

A Complexity Preserving Transformation from Jinja Bytecode to Rewrite Systems

Georg Moser

Institute of Computer Science,
University of Innsbruck, Austria
georg.moser@uibk.ac.at

Michael Schaper

Institute of Computer Science,
University of Innsbruck, Austria
michael.schaper@uibk.ac.at

February 5, 2013

We revisit known transformations from Jinja bytecode to rewrite systems from the viewpoint of runtime complexity. Suitably generalising the constructions proposed in the literature, we define an alternative representation of Jinja bytecode (JBC) executions as *computation graphs* from which we obtain a representation of JBC executions as *constrained rewrite systems*.

We prove non-termination and complexity preservation of the transformation. More precisely the runtime complexity of a given JBC program and the runtime complexity of the resulting rewrite system are related by a *linear factor*.

1. Introduction

In recent years research on complexity of rewrite systems has matured and a number of noteworthy results could be established. We give a quantitative assessment based on the annual competition of complexity analysers within TERMCOMP.¹ With respect to last year's run of TERMCOMP, we see a success rate of 38 % in the category *Runtime Complexity – Innermost Rewriting*. Note that the corresponding testbed is not restricted to polynomial runtime complexity in any way. With respect to a qualitative assessment we want to mention the very recent efforts to apply methods from linear algebra and

¹ <http://termcomp.uibk.ac.at/>.

automata theory to complexity [22]; recent efforts on adaption of the dependency pair method to complexity [15, 16, 26, 17] and the ongoing quest to incorporate compositionality [34, 4]. (See [23] for an overview in methods of complexity analysis of term rewrite systems.)

We are concerned with the question of applicability of these results to establish automated runtime complexity analysis of imperative programs, in particular of Jinja bytecode (JBC) programs. Jinja is a Java-like language that exhibits the core features of Java [33]. Its semantics is clearly defined and machine checked in the theorem prover Isabelle/HOL [18].

Our work is motivated by recent efforts to establish a non-termination preserving transformation from JBC programs to integer term rewrite system (ITRSs) [27, 8, 7, 6]. Building upon [32, 28] this method makes use of *termination graphs*. Termination graphs essentially are control-flow graphs using terms as abstraction domain, such that any evaluation is abstracted to a path in the graph. Then the termination graph becomes representable as an ITRS. As existing termination techniques can be adapted to ITRSs with relative ease, the termination graph method yields a competitive termination analysis of JBC programs.

Lifting this technique to an (automatable) complexity analysis is challenging. First, we have to establish that the transformation is sound from the viewpoint of complexity, ie., we have to establish *complexity preservation*: the runtime complexity of a given JBC program P is a (polynomial) function of the runtime complexity of a ITRS \mathcal{R} . Second, we have to adapt existing methods of complexity analysis to ITRSs. Thirdly, in order to make the analysis competitive, we have to provide *composability* of the obtained analysis, as is present in existing complexity tools for imperative programs [13, 12, 3, 1, 2, 35].

In this paper we are mainly concerned with the first challenge. We will come back to the two remaining challenges in the conclusions. Suitably generalising the constructions proposed in the literature, we define an alternative representation of JBC executions as *computation graphs* from which we obtain a representation of JBC executions as *constrained term rewrite systems* (*cTRSs* for short). CTRSs form a special type of rewrite systems that allow the formulation of conditions C over a theory T , such that a rule can only be used if the condition C is satisfied in T . Constraints are used to express relations on program variables. In our analysis we restrict to well-formed JBC programs that only make use of non-recursive methods and expect tree-shaped objects as input. Our main novel contributions are to extend results in the literature:

1. by basing our analysis directly on *Jinja bytecode* without relying on a boxed heap-model,
2. by providing a new graph-based representation of abstractions of JVM states that gives rise to a simplification and precision of *widening* of states,
3. by showing that any JBC program P subject to our analysis can be abstracted to a finite *computation graph*, which becomes easily representable as a cTRS,
4. by establishing complexity preservation of the transformation from JBC programs

P to constrained TRSs \mathcal{R} by a linear factor, ie., the runtime complexity with respect to P is linearly related to the runtime complexity with respect to \mathcal{R} ,

As a corollary to these results, we obtain a transformation from imperative programs to rewrite systems that is non-termination preserving, ie., any infinite evaluation of a JBC program P gives rise to an infinite derivation over the obtained rewrite system \mathcal{R} . We emphasise, that the proposed transformation is not directly automatable, but requires an extension by an external shape analysis (see for example [24, 5, 25]). Essentially exploiting the annotation technique initially proposed in [27] we have implemented a prototype of the approach to test its viability.

2. Related Work

The most overlap of our work is with earlier results reported for the termination graph method, in particular to the work by Otto et al. [27] and Brockschmidt et al. [8] that essentially share the same restrictions on the type of JBC programs analysed. The approach has been implemented in AProVE.² In contrast to these results our approach provides an alternative representation of abstract states that relies only on one simple form of annotations. In particular the tedious bookkeeping of annotations is avoided. Furthermore, while [27] (and follow-up work) rely on (unspecified) heuristics to guarantee a finite termination graph, we precise the notion of widening employed and show that finiteness of the computation graph can be guaranteed, even if widening is performed gently. On the downside our approach is not directly automatable as the translation to cTRSs relies on external analysis (or annotations) for cyclic or non-tree shaped data-structures.

Termination behaviour and complexity of such programs is studied by Albert et al. in [1, 2]. The approach employs program transformations to *constrained logic programs* and has been successfully implemented in the COSTA³ tool; it allows often surprisingly precise bounds on the resource usage and is not restricted to runtime complexity. Related work, targeting C programs has been reported by Alias et al. [3]. A theoretical limitation of the work is the focus on a path-length analysis of the heap, which does not provide the same detail as the term based abstraction presented here.

Gulwani and Zuleger study the *reachability-bound problem* that translates into bounds on the runtime complexity of a program [14]. In Zuleger et al. [35] the general methodology of [14] is refined by the use of *size-change abstraction*. The latter approach has been implemented in the tool LOOPUS. In connection with pathwise analysis and *contextualisation* size-change abstraction yields a powerful analysis. Both approaches rely on the use of (standard) invariant generation tools to link the bound program variable to the input. Furthermore, Gulwani et al. [12, 13] propose counter instrumentation of the code to a complexity analysis. Generally speaking, these methods are closely linked to Microsoft Research’s SPEED-project and study C programs. Our approach extends

² <http://aprove.informatik.rwth-aachen.de/>.

³ <http://costa.ls.fi.upm.es/>.

the use of transition systems by cTRSs, which theoretically form a strict extension. Furthermore, as our methods are rooted in rewriting we are not limited to the powers of invariant generation tools.

This paper is structured as follows. In Sections 3 and 4 we fix some basic notions to be used in the sequel. In particular, we give an overview over the Jinja programming language. Our notion of abstract states is presented in Sections 5 and 6, while computation graphs are proposed in Section 7. Section 8 introduces cTRSs and presents the transformation from computation graphs to rewrite systems. In Section 9 we briefly mention crucial design choices for our prototype implementation. Finally, in Section 10 we conclude.

3. Preliminaries

Let f be a mapping from A to B , denoted $f : A \rightarrow B$, then $\text{dom}(f) = \{x \mid f(x) \in B\}$ and $\text{rg}(f) = \{f(x) \mid x \in A\}$. Let $a \in \text{dom}(f)$. We define:

$$f\{a \mapsto v\} := \begin{cases} v & \text{if } x = a \\ f(x) & \text{otherwise} . \end{cases}$$

We usually use square brackets to denote a list. Further, $(::)$ denotes the cons operator, and $(@)$ is used to denote the concatenation of two lists.

Definition 3.1. A *directed graph* $G = (V_G, \text{Succ}_G, L_G)$ over the set \mathcal{L} of *labels* is a structure such that V_G is a finite set, the *nodes* or *vertices*, $\text{Succ}_G : V_G \rightarrow V_G^*$ is a mapping that associates a node u with an (ordered) sequence of nodes, called the *successors* of u . Note that the sequence of successors of u may be empty: $\text{Succ}_G(u) = []$. Finally $L_G : V_G \rightarrow \mathcal{L}$ is a mapping that associates each node u with its *label* $L_G(u)$. Let u, v be nodes in G , such that $v \in \text{Succ}_G(u)$ then there is an *edge* from u to v in G ; the edge from u to v is denoted as $u \rightarrow v$.

Definition 3.2. A structure $G = (V_G, \text{Succ}_G, L_G, E_G)$ is called *directed graph with edge labels* if $(V_G, \text{Succ}_G, L_G)$ is a directed graph over the set \mathcal{L} and $E_G : E_G \rightarrow \mathcal{L}$ is a mapping that associates each edge e with its *label* $E_G(e)$. Edges in G are denoted as $u \xrightarrow{\ell} v$, where $E_G(u \rightarrow v)$ and $u, v \in V_G$. We often write $u \rightarrow v$ if the label is either not important or is clear from context.

If not mentioned otherwise, in the following a *graph* is a directed graph with edge labels. Usually nodes in a graph are denoted by u, v, \dots possibly followed by subscripts. We drop the reference to the graph G from V_G , Succ_G , and L_G , ie., we write $G = (V, \text{Succ}, L)$ if no confusion can arise from this. Further, we also write $u \in G$ instead of $u \in V$.

Let $G = (V, \text{Succ}, L)$ be a graph and let $u \in G$. Consider $\text{Succ}(u) = [u_1, \dots, u_k]$. We call u_i ($1 \leq i \leq k$) the *i-th successor* of u (denoted as $u \xrightarrow{i}_G u_i$). If $u \xrightarrow{i}_G v$ for some i , then we simply write $u \rightarrow_G v$. A node v is called *reachable* from u if $u \xrightarrow{*}_G v$, where $\xrightarrow{*}_G$ denotes the reflexive and transitive closure of \rightarrow_G . We write $\xrightarrow{+}_G$ for $\rightarrow_G \circ \xrightarrow{*}_G$. A graph

G is *acyclic* if $u \xrightarrow{+}_G v$ implies $u \neq v$. We write $G \upharpoonright u$ for the subgraph of G reachable from u .

4. Jinja Bytecode

In this section, we give an overview over the Jinja programming language [18]. In particular we inspect the internal state of the Jinja Virtual Machine (JVM). We expect the reader to be familiar with the Java programming language.

Definition 4.1. A *value* in Jinja can be a Boolean, an integer, a reference (or address), the null reference (`null`), or the dummy value (`unit`).

We usually refer to (non-null) references as addresses. The dummy value `unit` is used for the evaluation of assignments (see [18]) and also used in the JVM to allocate uninitialised local variables.

Example 4.1. Figure 1 depicts a program defining a `List` object with the `append` method. Deviating from the notation employed by Klein and Nipkow in [18], we present Jinja code in a Java-like format.

```
class List{
  List next;
  int val;

  unit append(List ys){
    List cur = this;
    while(cur.next != null){
      cur = cur.next
    }
    cur.next = ys;
  }
}
```

Figure 1: The `List` program.

In preparation for the sequent sections, we reflect the structure and properties of JBC programs and the JVM.

Definition 4.2. A JBC *program* consists of a set of *class declarations*. Each class is identified by a *class name* and further consists of the name of its direct *superclass*, *field declarations* and *method declarations*. A field declaration is a pair of *field name* and *field type*. A method declaration consists of the *method name*, a list of *parameter types*, the *result type* and the *method body*.

A JBC method body is a triple of $(nat \times nat \times instructionlist)$. The two numbers represent the maximum size of the operand stack and the number of local variables, not including the `this` pointer and the parameters of the method, while *instructionlist* gives

a sequence of JBC instructions. The set of Jinja bytecode instructions is adapted for our needs and listed in Figure 2. We employ following conventions: Let n denote a natural number, i an integer, v a Jinja value, cn a class name, and mn a method name.

- | | |
|----------------------|---------------|
| • Load n | • IAdd |
| • Store n | • ISub |
| • Push v | • IfFalse i |
| • Pop | • CmpEq |
| • New cn | • Goto i |
| • Getfield fn cn | • CmpNeq |
| • Putfield fn cn | • CmpGeq |
| • Checkcast cn | • BAnd |
| • Invoke mn | • BOr |
| • Return | • BNot |

Figure 2: The Jinja bytecode instruction set.

Definition 4.3. A *(JVM) state* is a pair consisting of the *heap* and a list of *frames*. Let \prec denote the strict subclass relation and \preceq its reflexive closure. A *heap* is a mapping from *addresses* to *objects*, where an object is a pair $(cn, ftable)$ such that:

- cn denotes the *class name*, and
- $ftable$ denotes the *fieldtable*, ie., a mapping from (cn', fn) to values, where fn is a *field name* and cn' is a (not necessarily proper) superclass of cn , ie., $cn \preceq cn'$.

A *frame* represents the environment of a method and is a quintuple (stk, loc, cn, mn, pc) , such that:

- stk denotes the *operation stack*, ie., an array of values,
- loc denotes the *registers*, ie., an array of values,
- cn denotes the *class name*,
- mn denotes the *method name*, and
- pc is the *program counter*.

Let stk (loc) denote the operation stack (registers) of a given frame. Typically the structure of loc is as follows: the 0^{th} register holds the *this*-pointer, followed by the parameters and the local variables of the method. Uninitialised registers are preallocated with the dummy value `unit`. We denote the entries of stk (loc), by $stk(i)$ ($loc(i)$) for $i \in \mathbb{N}$ and write $\text{dom}(stk)$ ($\text{dom}(loc)$) for the set of indices of the array stk (loc). Often there is no need to separate between the local variables of a Jinja program and the registers in a JBC program. Hence we use registers and local variables interchangeably.

$$\text{IAdd} \quad \frac{(heap, (i_2 :: i_1 :: stk, loc, cn, mn, pc) :: frms)}{(heap, ((i_2 + i_1) :: stk, loc, cn, mn, pc + 1) :: frms)}$$

Figure 3: The IAdd bytecode instruction.

Figure 3 illustrates the one-step execution of the IAdd bytecode instruction. We have extended the original set of instructions by some standard operations on values, taking ideas from Jinja with Threads into account [20, 21]. The semantics of all employed JBC instruction can be found in the Appendix.

Example 4.2. Consider the List program from Example 4.1. Figure 4 depicts the corresponding bytecode program, resulting from the compilation rules in [18]. In the following we name the registers 0,1, and 2 as *this*, *ys*, and *cur*, respectively.

Class:	
Name: List	Bytecode:
Classbody:	00: Load 0
Superclass: Object	01: Store 2
Fields:	02: Push unit
List next	03: Pop
int val	04: Load 2
Methods:	05: Getfield next List
Method: unit append	06: Push null
Parameters:	07: CmpNeq
List ys	08: IfFalse 7
Methodbody:	09: Load 2
MaxStack:	10: Getfield next List
2	11: Store 2
MaxVars:	12: Push unit
1	13: Pop
	14: Goto -10
	15: Push unit
	16: Pop
	17: Load 2
	18: Load 1
	19: PutField next List
	20: Push unit
	21: Return

Figure 4: The bytecode for the List program.

The bytecode verifier established in [18] ensures following properties: All bytecode instructions are provided with arguments of the expected type. No instruction tries to get a value from the empty stack, nor puts more elements on the stack or access more registers than specified in the method. The program counter is always within the code array of the method. All registers except from the register storing *this* must be first

written to before accessed. Furthermore the verifier ensures that for states with equal program counter the size of the stack is of equal length. Moreover, the list of registers is of fixed length. The compiler presented in [18] transforms a well-formed Jinja program into a well-formed JBC program. A JBC program that passes the bytecode verification is again called *well-formed*.

While the set of instruction used here are a (slight) extension of the minimalistic set considered in [18], this notion of well-formedness is still applicable, as all considered extensions are present in Jinja with Threads [20, 21]. In the following we consider Jinja programs and JBC programs to be well-formed. To ease readability we do not consider exception handling, that is, an exception yields immediate termination of the program. This is not a restriction of our analysis, as it could be easily integrated, but complicates matters without gaining additional insight.

Let P be a program and let s and t be states. Then we denote by $P: s \xrightarrow{\text{jvm}}_1 t$ the one-step transition relation of the JVM. If there exists a (normal) evaluation of s to t , we write $P: s \xrightarrow{\text{jvm}} t$.

We define the *runtime* of a JVM for a given normal evaluation $P: s \xrightarrow{\text{jvm}} t$ as the number of single-step executions in the course of the evaluation from s to t .

Definition 4.4. We define the *runtime complexity* with respect to P as follows:

$$\text{rcjvm}(n) := \max\{m \mid P: \text{start} \xrightarrow{\text{jvm}} t \text{ holds such that the runtime is } m, \\ \text{start is an initial state, and } \|\text{start}\| \leq n \} .$$

Here $\|\cdot\|$ denotes a suitable size measure for states; the measure is made precise below.

5. Abstract States

In this section, we introduce *abstract states* as generalisations of JVM states. The intuition being that abstract states represent sets of states in the JVM. The idea of abstracting JVM states in this way is due to Otto et al. [27]. However, our presentation crucially differs from [27] (and also from follow-up work in the literature) as we employ an implicit representation of sharing that makes use of graph morphisms, rather than the explicit sharing information proposed in [27, 8, 7, 6]. Furthermore, abstract states as defined below are a straightforward generalisation of JVM states as defined in [18]. This circumvents an additional transformation step as presented in [8].

Definition 5.1. We extend Jinja expressions by countable many abstract variables X_1, X_2, X_3, \dots , denoted by x, y, z, \dots . An *abstract variable* may either abstract an object, an integer or a Boolean value.

In denoting abstract variables typically the name is of less importance than the type, that is we denote an abstract variable for an object of class cn , simply as cn , while abstract integer or Boolean variables are denoted as int , and $bool$, respectively. The (strict) subclass relation $(\prec) \preceq$ is extended in the natural way to abstract variables for classes.

Definition 5.2. An *abstract value* is either a value (cf. Definition 4.1), or an abstract Boolean or integer value. As in the JVM, only (abstract) objects can be shared. In particular note that abstract variables for objects are only referenced via the heap.

The next definition abstracts the heap of a JVM through the use of abstract variables and values.

Definition 5.3. An *abstract heap* is a mapping from *addresses* to *abstract objects*, where an abstract object is either a pair $(cn, ftable)$ or an abstract variable. We define (partial) projection functions cl and ft as follows:

$$cl(obj) := \begin{cases} cn & \text{if } obj \text{ is an object and } obj = (cn, ftable) \\ cn & \text{if } obj \text{ is an abstract variable of type } cn \end{cases}$$

$$ft(obj) := \begin{cases} ftable & \text{if } obj \text{ is an object and } obj = (cn, ftable) \\ \text{undefined} & \text{otherwise} \end{cases}.$$

Abstract frames are defined like frames of the JVM, but registers and operand stack of an abstract frame store abstract values. Furthermore, we define *annotations* of addresses in a state s , denoted as iu . Formally, the annotations are pairs $p \neq q$ of addresses, where $p, q \in heap$ and $p \neq q$. The intuition of iu is to express that for $p \neq q \in iu$, we disallow sharing of these addresses in states represented by the state s .

Definition 5.4. An *abstract state* $s = (heap, frms, iu)$ is a triple consisting of an abstract heap $heap$, a list of abstract frames $frms$, and a set of annotations iu . Furthermore, we demand that all addresses in $heap$ are reachable from local variables or stack entries in the list of frames $frms$.

When depicting states, we replace stack and register indices by intuitive names, denoted in roman font. Furthermore, we make use of the following conventions: we use an italic font (and lower-case) to describe abstract variables and a sans serif (and upper-case) to depict class names.

Example 5.1 (continued from Example 4.1). Consider the `List` program from Example 4.1 together with the well-formed JBC program depicted in Figure 4. Consider the state A depicted below:

04	$\epsilon \mid this = o_1, ys = o_2, cur = o_1$
A	$o_1 = \text{List}(\text{List.val} = int, \text{List.next} = o_3)$ $o_2 = list, o_3 = list$

The operation stack in A is empty. The registers *this* and *cur* contain the same address o_1 and *ys* is mapped to o_2 . In the heap o_1 is mapped to an object of type `List` whose value is abstracted to *int* and whose next element is referenced by o_3 . It is not difficult to see that A forms an abstraction of any JVM state obtained at instruction **04** in the `List` program (if *this* initially references a non-empty list) before any iteration of the `while`-loop. Furthermore, consider the following state B :

04	$\epsilon \mid \text{this} = o_1, \text{ys} = o_2, \text{cur} = o_3$ $o_1 = \text{List}(\text{List.val} = \text{int}, \text{List.next} = o_3)$ $o_2 = \text{list}, o_4 = \text{list}$
B	$o_3 = \text{List}(\text{List.val} = \text{int}, \text{List.next} = o_4)$

Again it is not difficult to see that B abstracts any JVM state obtained if exactly one iteration of the loop has been performed.

In the presence of abstract variables an abstract state represents a set of JVM states. As it will always be clear from the context, whether we refer to an abstract or a JVM state, we drop the qualifier “abstract” in the following.

Definition 5.5. We define a preorder on abstract (non-address) values and objects; $v \trianglelefteq w$ holds if $v = w$ or

- $v = \text{unit}$ and $w = \text{null}$ or w is an abstract variable of type int , bool , or cn ,
- $v = \text{null}$ and w is cn ,
- $v \in \mathbb{Z}$ or v an abstract integer and w an abstract integer,
- $v \in \{\text{true}, \text{false}\}$ or v an abstract Boolean and w an abstract Boolean, or
- $\text{cl}(v) = \text{cn}'$ and w is an abstract variable of type cn such that $\text{cn}' \preceq \text{cn}$.

We write $w \trianglerighteq v$, if $v \trianglelefteq w$.

The presence of abstract variables in states allows to abstract away certain details of a given state t . This intuition is made precise in the next definition. Let $|stk|$, $|loc|$ denote the maximum size of the operand stack and the number of variables respectively. We make use of the following abbreviation: $w \trianglerighteq_m v$ if either $w \trianglerighteq v$ or v, w are references and we have $v = m(w)$, where m denotes a mapping on references.

Definition 5.6. Let $s = (\text{heap}, \text{frms}, \text{iu})$ be a state with $\text{frms} = [\text{frm}_1, \dots, \text{frm}_k]$ and $\text{frm}_i = (\text{stk}_i, \text{loc}_i, \text{cn}_i, \text{mn}_i, \text{pc}_i)$. Furthermore let $t = (\text{heap}', \text{frms}', \text{iu}')$ be a state with $\text{frms}' = [\text{frm}'_1, \dots, \text{frm}'_k]$ and $\text{frm}'_i = (\text{stk}'_i, \text{loc}'_i, \text{cn}'_i, \text{mn}'_i, \text{pc}'_i)$.

Then s is an *abstraction* of t (denoted as $s \sqsupseteq t$) if the following conditions hold:

1. for all $1 \leq i \leq k$: $\text{pc}_i = \text{pc}'_i$, $\text{cn}_i = \text{cn}'_i$, and $\text{mn}_i = \text{mn}'_i$,
2. for all $1 \leq i \leq k$: $\text{dom}(\text{stk}_i) = \text{dom}(\text{stk}'_i)$ and $\text{dom}(\text{loc}_i) = \text{dom}(\text{loc}'_i)$,
3. there exists a mapping $m: \text{dom}(\text{heap}) \rightarrow \text{dom}(\text{heap}')$ such that
 - for all $1 \leq i \leq k$, $1 \leq j \leq |\text{stk}_i|$: $\text{stk}_i(j) \trianglerighteq_m \text{stk}'_i(j)$,
 - for all $1 \leq i \leq k$, $1 \leq j \leq |\text{loc}_i|$: $\text{loc}_i(j) \trianglerighteq_m \text{loc}'_i(j)$,
 - for all $a \in \text{dom}(\text{heap})$: $\text{heap}(a) \trianglerighteq_m \text{heap}'(m(a))$,

- for all $a \in \text{dom}(\text{heap})$, such that $\text{ft}(\text{heap}(a))$ is defined and for all $1 \leq i \leq \ell$:
 $f(cn_i, id_i) \supseteq_m f'(cn'_i, id_i)$,
where $f := \text{ft}(\text{heap}(a))$ with $\text{dom}(f) = \{(cn_1, id_1), \dots, (cn_\ell, id_\ell)\}$ and
 $f' := \text{ft}(\text{heap}'(m(a)))$ with $\text{dom}(f') = \{(cn_1, id_1), \dots, (cn_\ell, id_\ell)\}$.

4. finally, we have $iu' \supseteq m^*(iu)$.

Example 5.2 (continued from Example 5.1). Consider the states A and B described in Example 5.1. For the state S depicted below we obtain that $A \sqsubseteq S$ and $B \sqsubseteq S$, ie., S forms an abstraction of both states.

04	$\epsilon \mid \text{this} = o_1, \text{ys} = o_2, \text{cur} = o_4$
	$o_1 = \text{List}(\text{List.val} = \text{int}, \text{List.next} = o_3)$
	$o_2 = \text{list}, o_3 = \text{list}, o_5 = \text{list}$
S	$o_4 = \text{List}(\text{List.val} = \text{int}, \text{List.next} = o_5)$

Strictly speaking Definition 5.6 does not apply to JVM states as clearly the latter do not contain annotations. However, to simplify the notation, we identify JVM states with abstract states where all references are marked as different in the annotation iu . Thus the relation \sqsubseteq becomes applicable to relate program states and their abstractions.

While the above form of representing states allows for a succinct presentation, it is more natural to conceive the heap (and conclusively a state) as a graph. In the next section, we make this intuition precise.

6. Graph-Based Representation of States

Let s be a state and let heap denote the heap of s . We propose a graph-based representations of heap and state s called *heap graph* and *state graph* respectively. This representation makes use of a set $\mathcal{I}_{\text{heap}}$ of *implicit reference*. Suppose a is an address on heap , cn a classname, and id a field identifier. Furthermore, suppose $\text{ftable} = \text{ft}(\text{heap}(a))$ is defined and $\text{ftable}((cn, id)) = \text{val}$, such that val not an address. Then we say the triple (a, cn, id) is an *implicit reference* for val ; the set $\mathcal{I}_{\text{heap}}$ collects all implicit references of heap .

Definition 6.1. Let heap denote the heap. We represent heap as a directed graph with edge labels $H = (V_H, \text{Succ}_H, L_H, E_H)$, where the nodes, the successor relations and the

labelling functions are defined as follows:

$$\begin{aligned}
V_H &:= \text{dom}(\text{heap}) \cup \text{dom}(\mathcal{I}_{\text{heap}}) \\
\text{Succ}_H(u) &:= \begin{cases} [f^*(u, (cn_1, id_1)), \dots, f^*(u, (cn_k, id_k))] & \text{if } u \text{ is an address,} \\ & \text{ft}(\text{heap}(u)) = f, \text{ dom}(f) = \\ & \{(cn_1, id_1), \dots, (cn_k, id_k)\} \\ \emptyset & \text{otherwise .} \end{cases} \\
L_H(u) &:= \begin{cases} \text{cl}(\text{heap}(u)) & \text{if } u \text{ is an address} \\ val & \text{if } u \text{ is an implicit reference for } val . \end{cases} \\
E_H(u \rightarrow v) &:= \begin{cases} (cn, id) & \text{if } u \text{ is an address, } \text{ft}(\text{heap}(u)) =: f \text{ and } f^*(u, (cn, id)) = v \\ \emptyset & \text{otherwise .} \end{cases}
\end{aligned}$$

Here $f^*(u, (cn, id)) := f((cn, id))$, if $f((cn, id))$ is an address and $f^*(u, (cn, id)) := (u, cn, id)$ otherwise, where $(u, cn, id) \in \mathcal{I}_{\text{heap}}$.

We usually confuse the heap with its representation as graph. In particular we call a value val reachable from an address a in heap , if there exists a path from a to val in the heap graph of heap . Based on the graph representation of heap , we represent s as a state graph S .

Let $s = (\text{heap}, \text{frms}, iu)$ be a state and let $\text{frms} = [\text{frm}_1, \dots, \text{frm}_k]$, such that $\text{frm}_i = (\text{stk}_i, \text{loc}_i, \text{cn}_i, \text{mn}_i, \text{pc}_i)$. We define the set $\text{Stk}(s) := \{(\text{stk}, i, j) \mid 1 \leq i \leq k, 1 \leq j \leq |\text{stk}_i|\}$ that collects all *stack indices*. Similar we define the set of *register indices*: $\text{Loc}(s) := \{(\text{loc}, i, j) \mid 1 \leq i \leq k, 1 \leq j \leq |\text{loc}_i|\}$. If s is clear from context, we write Stk (Loc) instead of $\text{Stk}(s)$ ($\text{Loc}(s)$). We extend the set $\mathcal{I}_{\text{heap}}$ to cover also non-address values stored in the stack or registers. For this it suffices to extend $\mathcal{I}_{\text{heap}}$ by a disjoint copy of $\text{Stk}(s) \cup \text{Loc}(s)$. The set of implicit references with respect to s is denoted as \mathcal{I}_s . The copy of a stack or register index in \mathcal{I}_s is called its *implicit reference*.

Definition 6.2. Let $s = (\text{heap}, \text{frms}, iu)$ be a state and let H denote the heap graph of heap . Furthermore, let $(\text{stk}, i, j) \in \text{Stk}$ and let $(\text{loc}, i', j') \in \text{Loc}$. We write $os_{(i,j)}$ and $l_{(i',j')}$ to name the indices of the operation stack and registers.

We define the *state graph of s* as 5-triple $S = (V_S, \text{Succ}_S, L_S, E_S, iu)$, where the first four components denote a directed graph with edge labels and iu denotes a set of annotations. The nodes, the successor relation, and the labelling function of the directed

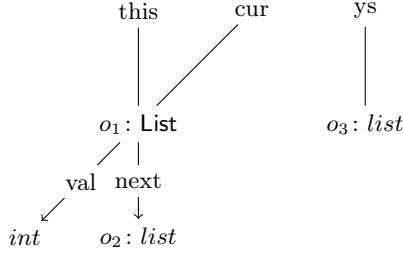


Figure 5: Abstract State A

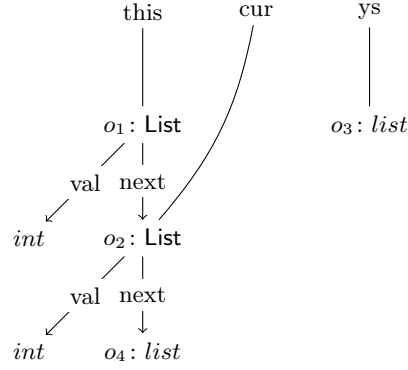


Figure 6: Abstract State B

graph are defined as follows:

$$\begin{aligned}
 V_S &:= Stk \cup Loc \cup V_H \cup \mathcal{I}_s \\
 Succ_S(u) &:= \begin{cases} [stk_i^*(j)] & \text{if } u = (stk, i, j) \in Stk \\ [loc_i^*(j)] & \text{if } u = (loc, i, j) \in Loc \\ Succ_H(u) & \text{if } u \in V_H . \end{cases} \\
 L_S(u) &:= \begin{cases} os_{(i,j)} & \text{if } u = (stk, i, j) \in Stk \\ l_{(i,j)} & \text{if } u = (loc, i, j) \in Loc \\ L_H(u) & \text{if } u \in V_H \\ val & \text{if } u \text{ is an implicit reference for } val . \end{cases} \\
 E_S(u \rightarrow v) &:= \begin{cases} E_H(u \rightarrow v) & \text{if } u, v \in H \\ \emptyset & \text{otherwise .} \end{cases}
 \end{aligned}$$

Here $stk_i^*(j)$ and $loc_i^*(j)$ is defined like f^* as introduced in Definition 6.1, ie., $stk_i^*(j) := stk_i(j)$ ($loc_i^*(j) := loc_i(j)$), if $stk_i(k)$ ($loc_i(j)$) is an address and $stk_i^*(j)$ is defined as the implicit reference of (stk, i, j) ; similarly for $(loc_i^*(j))$.

We often confuse a state s and its representation as a state graph S . In presenting state graphs, we indicate references, but do not depict implicit references. The graph-based representation S provides a much better intuition about the notion of instance of abstraction of s , cf. Definition 5.6. In particular it will turn out that Definition 5.6 amounts to a variant of graph morphism on two different graph representation of states.

Example 6.1 (continued from Example 5.2). Consider the states A , B , and S presented in Example 5.2. The state graph of A and B are given in Figure 5 and Figure 6, respectively. The state graph of the abstraction S is depicted in Figure 7.

The *size of a state* is defined on a *per-reference* basis, which unravels sharing.

Definition 6.3. Let s be a state and let S be its state graph. Let u, v be nodes in S . We write $u \xrightarrow{*} v$, if there exists a simple path P in S from u to v . Note that P must not contain cycles. Then the size of a stack or register index u (denoted as $\|u\|$) is defined as follows:

$$\|u\| := \sum_{u \xrightarrow{*} v} \|L_S(v)\|,$$

where $\|l\|$ is $\mathbf{abs}(l)$ if $l \in \mathbb{Z}$, otherwise 1. (As usual $\mathbf{abs}(z)$ denotes the absolute value of the integer z .) Then the *size* of s is the sum of all sizes of stack or register indices in S .

Based on the relation \triangleright , cf. Definition 5.5, we introduce the following variant of graph morphism, called *state homomorphism*.

Definition 6.4. Let S and T be state graphs of states s and t , respectively. A *state homomorphism* from S to T (denoted $m: S \rightarrow T$) is a function $m: V_S \rightarrow V_T$ such that

1. $Stk(s) = Stk(t) =: Stk$ and $Loc(s) = Loc(t) =: Loc$,
2. for all $u \in S$ and $u \in Stk \cup Loc$, $u = m(u)$ and $L_S(u) = L_T(m(u))$,
3. for all $u \in S \setminus (Stk \cup Loc)$, $L_S(u) \triangleright L_T(m(u))$,
4. for all $u \in S$ such that $Succ_S(u) \neq \emptyset$, $m^*(Succ_S(u)) = Succ_T(m(u))$, and
5. for all $u \xrightarrow{\ell} v \in S$ and $m(u) \xrightarrow{\ell'} m(v) \in T$, $\ell = \ell'$.

We use m^* to denote the lifting of m to non-empty lists or sets: $m([u_1, \dots, u_k]) = [m(u_1), \dots, m(u_k)]$.

If no confusion can arise we refer to a state homomorphism simply as *morphism*. It is easy to see that the composition $m_1 \circ m_2$ of two morphisms m_1, m_2 is again a morphism.

Lemma 6.1. Let $s = (heap, frms, iu)$ and $t = (heap', frms', iu')$ be states, whose state graphs are denoted by S and T respectively. Suppose the program counters, the class and method names of all frames coincide. Furthermore suppose that $iu' \supseteq m^*(iu)$ and assume there exists a morphism $m: S \rightarrow T$; then $s \sqsubseteq t$.

Proof. Straightforward. □

It is an easy consequence of Lemma 6.1 and the composability of morphism that the instance relation \sqsubseteq is transitive. Hence the relation \sqsubseteq is a preorder. In the following we aim to provide a mechanism to widen two distinct abstract states.

Definition 6.5. Let s and s' be states, such that there exists an abstraction t of s and s' . We call t the *join* of s and s' , denoted as $s \sqcup s'$, if t is a least upper bound of $\{s, s'\}$ with respect to the preorder \sqsubseteq .

In order to prove the existence of $s \sqcup s'$, we identify invariants of abstractions. Let $S = (V_S, Succ_S, L_S, E_S, iu_S)$ and $S' = (V_{S'}, Succ_{S'}, L_{S'}, E_{S'}, iu_{S'})$ be the two state graphs of state s and s' , respectively. Furthermore let t be an abstraction of s and s' and $T = (V_T, Succ_T, L_T, E_T, iu_T)$ its state graph. By definition we have the following properties:

1. Let Stk (Loc) collect the stack (register) indices of state s . As $s \sqsubseteq t$, Stk (Loc) coincide with the set of stack (register) indices of t . Similarly for s' and thus $V_T \supseteq Stk \cup Loc$.
2. For any node $u \in T$ there exists uniquely defined nodes $v \in V_S$, $w \in V_{S'}$ such that $L_S(v) \trianglelefteq L_T(u)$, $L_{S'}(w) \trianglelefteq L_T(u)$. We say the nodes v and w correspond to u .
3. For any node $u \in T$ and any successor u' of u in T there exists a successor v' (w') in S (S') of the corresponding node v (w) in S (S'). Furthermore v' and w' correspond to u' .
4. For any edge $u \xrightarrow{\ell} u' \in T$ such that v (w) corresponds to u in S (S') there is an edge $v \xrightarrow{k} v' \in S$ and an edge $w \xrightarrow{k'} w' \in S'$ such that $\ell = k = k'$.
5. For any annotation $u \neq u' \in iu_T$ there exists $v \neq v'$ in iu_S and $w \neq w'$ in $iu_{S'}$ such that v (v') and w (w') correspond to u (u').

In order to construct an abstraction t of s and s' we use the above properties as invariants and define its state graph T by iterated extension. We define T^0 by setting $V_{T^0} := Stk \cup Loc$. Due to Property 1 these nodes exist in S' as well. The labels of stack or register indices trivially coincide in S and S' , cf. Definition 6.4. Thus we set L_{T^0} accordingly. Furthermore we set $Succ_{T^0} = E_{T^0} = iu_{T^0} := \emptyset$. Then T^0 satisfies Properties 1–5.

Suppose state graph T^n has already been defined such that the Properties 1–5 are fulfilled. In order to update T^n , let $u \in V_{T^n}$ such that v and w correspond to u . Suppose $v \xrightarrow{k} v' \in S$ and $w \xrightarrow{k'} w' \in S'$ such that there is no node u' in T^n where v' and w' correspond to u' . Let u' denote a node fresh to T^n . We define $V_{T^{n+1}} := V_{T^n} \cup \{u'\}$ and establish Property 2 by setting $L_{T^{n+1}}(u')$ such that $L_S(v') \sqsubseteq L_{T^{n+1}}(u')$ and $L_{S'}(w') \sqsubseteq L_{T^{n+1}}(u')$ where $L_{T^{n+1}}(u')$ is as concrete as possible. If we succeed, we fix that v' and w' correspond to u' . It remains to update $iu_{T^{n+1}}$ suitably such that Property 5 is fulfilled. If this also succeeds Properties 1–5 are fulfilled for T^{n+1} . On the other hand, if no further update is possible we set $T := T^n$. By construction T is an abstraction of S and S' and indeed represents $s \sqcup s'$.

Definition 6.6. As \sqcup is associative and commutative, we can extend the binary operation \sqcup to define the least upper bound of a set of states \mathcal{T} , denoted as $\sqcup \mathcal{T}$. We call the abstraction of the states in \mathcal{T} by $\sqcup \mathcal{T}$ *widening*.

Example 6.2 (continued from Example 5.2). Consider the states A , B , and S described in Example 5.2. In Figure 7 an abstraction of A and B is given. In particular, abstraction S results of the construction defined above, ie., $S = \sqcup \{A, B\}$.

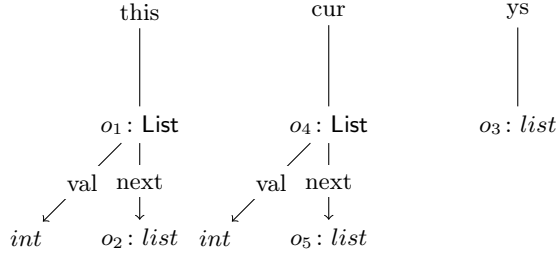


Figure 7: Abstraction S

Let P be a fixed JBC program. In the next section we propose computation graphs as finite representations of all possible JVM states of P . The nodes of computation graphs are abstract states.

7. Computation Graphs

In this section, we define *computation graphs* as *finite* representations of all possible representation of the program flow of a given JBC program. Computation graphs are strongly related to *termination graphs* as proposed by Otto et al. [27] and in particular Brockschmidt et al. [8].

Above, we have already restricted our attention to well-formed JBC programs P using the expressions and instructions defined in Section 4. For the proposed static analysis of these programs we make the following additional restrictions. First, we restrict to *non-recursive* methods. Note that the abstract states defined above (cf. Definition 5.4) can in principle express recursive methods, but for recursive methods, we cannot use the below proposed construction to obtain *finite* computation graphs, as the graphs defined in Definition 7.4 cannot handle unbounded list of frames. Second, in the final transformation to cTRSs, we abstract non tree-shaped objects, ie., for example whenever P creates a cyclic object, we represent it by an abstract variable of the corresponding class. Note that our notion of abstract states, and thus also computation graphs can express non-tree shaped, and even cyclic objects, but we can only represent tree-shaped objects as terms. Let P denote a well-formed JBC program based on data in tree-shaped form that only makes use of non-recursive methods. P is fixed for the remainder of this paper.

Suppose two addresses p on q in a state s could potentially be shared in an instance of s . Then we call p and q *unifiable*.

Definition 7.1. Let $s = (\text{heap}, \text{frms}, \text{iu})$ be a state and let p, q denote distinct addresses in *heap* such that $p \neq q \notin \text{iu}$. Then we say p and q are *unifiable* (denoted as $p \stackrel{?}{=} q$) if there is a JVM state t reachable in P from some initial state *start* and a morphism $m: s \rightarrow t$, such that $m(p) = m(q)$.

Figure 3 presents the single-step execution of the **IAdd** instruction (see [18] for the rest). Based on these instructions, and actually mimicking them quite closely, we define

how abstract states are symbolically evaluated. In Figure 8 we have worked out the cases for the instructions **New**, **Putfield**, **IfFalse**, **IAdd**, **CmpEq**, and **BAnd**. We follow the notation used in Figure 3 above. The other cases are left to the reader.

Some comments: The symbolic instruction **New** cn' creates a new entry in the heap such that a is mapped to a new abstract variable of type cn' . Formally, let a be a new address and x a new abstract variable of type cn' . We define the mapping $heap'$ such that $\text{dom}(heap') = \text{dom}(heap) \cup \{a\}$. Let v be a value and a be an address such that $heap(a) = (cn'', ftable)$. The application of the instruction **Putfield** fn cn' is only possible if there exists no address $p \in heap$ such that $a \stackrel{?}{=} p$. For the **IAdd** instruction, we introduce a new abstract integer i_3 and the side-condition $i_1 + i_2 = i_3$. Finally, the **CmpEq** splits into different cases, depending on the status of the compared values. Note that **CmpEq** does only perform an equality check on Jinja values and due to the abstraction on the heap we have to be careful when comparing addresses:

1. Let val_1 and val_2 be addresses. If the addresses of val_1 and val_2 are the same then the test evaluates to **true**. Otherwise, we have to check if val_1 and val_2 unify and perform a unsharing refinement according to Definition 7.3 if necessary.
2. Wlog. let val_1 be an address and val_2 be **null**. If $heap(val_1) = obj$ and $cl(obj) = cn$, we perform a instance refinement according to Definition 7.2 on val_1 .
3. If val_1 and val_2 are concrete non-address Jinja values, then the test ($val_1 = val_2$) can be directly executed and the symbolic execution equals the instruction on the JVM.
4. If val_1 and val_2 are abstract Boolean or integer variables, then we introduce a new Boolean variable b_3 and the side condition $(val_1 = val_2) \equiv b_3$. Figure 8 only shows the latter case.

In addition to symbolic evaluations, we define refinement steps on abstract states s if the information given in s is not concrete enough to execute a given instruction. Following [8] we make use of *class instance* and *unsharing*. Note, that it will be a consequence of our definitions that for any refinement s of a state t , we have $s \sqsubseteq t$.

Definition 7.2. Let $s = (heap, frms, iu)$ be a state and let a be an address such that $cl(heap(a)) = cname$. Suppose $subclasses := \{cn \mid cn \preceq cname\}$ and let $cn \in subclasses$. Furthermore, suppose $(cn_1, id_1), \dots, (cn_n, id_n)$ denote fields of cn (together with the defining classes).

We obtain the following two refinement steps, where the second takes care of the case, where abstract variable at address a is replaced by the **null** pointer.

$$\frac{(heap, frms, iu)}{(heap\{a \mapsto (cn, ftable_1)\}, frms, iu)} \qquad \frac{(heap, frms, iu)}{(heap_2, frms_2, iu)}.$$

Here $ftable_1((cn_i, id_i)) := v_i$ such that the type of the abstract variable v_i is defined in correspondence to the definition of cn_i . On the other hand we set $heap_2(frms_2)$ equal to $heap(frms)$, but $a \notin \text{dom}(heap_2)$ and all occurrences of a are replaced by **null**.

New cn'	$\frac{(heap, (stk, loc, cn, mn, pc) :: frms, iu)}{(heap' \{a \mapsto x\}, (a :: stk, loc, cn, mn, pc + 1) :: frms, iu)}$	
Putfield $fn\ cn'$	$\frac{(heap, (v :: a :: stk, loc, cn, mn, pc) :: frms, iu)}{(heap \{a \mapsto (cn'', ftable')\}, (stk, loc, cn, mn, pc + 1) :: frms, iu)}$	
IAdd	$\frac{(heap, (i_2 :: i_1 :: stk, loc, cn, mn, pc) :: frms, iu)}{(heap, (i_3 :: stk, loc, cn, mn, pc + 1) :: frms, iu)}$	$i_1 + i_2 = i_3$
IfFalse i	$\frac{(heap, (false :: stk, loc, cn, mn, pc) :: frms, iu)}{(heap, (stk, loc, cn, mn, pc + i) :: frms, iu)}$	
	$\frac{(heap, (true :: stk, loc, cn, mn, pc) :: frms, iu)}{(heap, (stk, loc, cn, mn, pc + 1) :: frms, iu)}$	
CmpEq	$\frac{(heap, (val_2 :: val_1 :: stk, loc, cn, mn, pc) :: frms, iu)}{(heap, (b_3 :: stk, loc, cn, mn, pc + 1) :: frms, iu)}$	$(val_1 = val_2) \equiv b_3$
BAnd	$\frac{(heap, (b_2 :: b_1 :: stk, loc, cn, mn, pc) :: frms, iu)}{(heap, (b_3 :: stk, loc, cn, mn, pc + 1) :: frms, iu)}$	$b_2 \wedge b_1 \equiv b_3$

Figure 8: Symbolic evaluations of Jinja bytecode instructions

```

class A{
  unit m(){unit}
}
class B extends A{
  unit m(){while(true)}
}

class C{
  unit call(A a){a.m()}
  main(){
    C c = new C();
    c.call(new B());
  }
};

```

Figure 9: All subclasses need to be considered.

Example 7.1. In Figure 9 we present an example detailing the need for the given definition of class instantiation. Here class B overrides method `m` inherited from class A. We only know the static type of the parameter when analysing method `call(A a)`. Method `call(A a)` accepts any instances of class A or any instances of a subclass of A as parameter. In particular any instance of class B. Due to the overridden method `call(A a)` does not terminate for instances of class B.

Definition 7.3. Let $s = (heap, frms, iu)$ be state, let $S = (V, Succ, L, E, iu)$ denote its state graph, and let p and q denote addresses in $heap$ such that $p \stackrel{?}{=} q$, that is, p and q potentially represent the same object. We obtain the following refinement steps: The first case forces these addresses to be distinct. The second case substitutes all occurrences of q with p .

$$\frac{(heap, frms, iu)}{(heap, frms, iu \cup \{p \neq q\})} \qquad \frac{(heap, frms, iu)}{(heap', frms', iu)},$$

where $heap'$ ($frms'$) is equal to $heap$ ($frms$) with all occurrences of q replaced by p .

Let s and t be abstract states such that s' is obtained from s due to a symbolic evaluation (cf. Figure 8) a case distinction (Definition 7.2) or an unsharing step (Definition 7.3). Then we say t is obtained from s by an *abstract computation*.

Definition 7.4. A *computation graph* $G = (V_G, E_G)$ is a directed graph with edge labels, where V_G are abstract states and $s \xrightarrow{\ell} t \in E_G$ if either t is obtained from s by an abstract computation or s is an instance of t . Furthermore, if there exists a constraint C in the symbolic evaluation, then $\ell := C$. For all other cases $\ell := \emptyset$. We say that G is the computation graph of program P if for all initial states $start$ of P there exists an abstract state $I \in G$ such that $start \sqsubseteq I$.

Example 7.2 (continued from Example 4.1). Consider the `List` program from Example 4.1 and the corresponding bytecode from Example 4.2. Figure 10 illustrates the computation graph of `append`. For the sake of readability we omit the *val* field of the list. Note that this graph is not complete, ie., we omit some intermediate states and do not illustrate all refinement cases.

First, consider the initial node I . It is easy to see that I is an abstraction of all concrete initial states. Nodes A , B and S correspond to the situation described in Example 5.1 and Example 5.2. That is, node A is obtained after assigning *cur* to *this* before any iteration of the loop, node B is obtained after exactly one iteration of the loop and node $S = \bigsqcup \{A, B\}$. We usually do not consider intermediate iterations. That is why node B is illustrated by a dashed border.

After pushing the reference of *cur.next* and `null` onto the operand stack, we reach node C . At `pc = 7` we want to compare the reference of *cur.next* with `null`. But, *cur.next* is not concrete. Therefore, a class instance refinement is performed, yielding nodes C' and C'' .

First, we consider that *cur.next* is not `null`, but references an arbitrary instance. This is illustrated in node C' . The step from C' to D is trivial. Let *id* denote the identity function and $m = id(V_S)$. Then $m\{o_4 \mapsto o_5, o_5 \mapsto o_6\}$ is a morphism from S to D .

Therefore, D is an instance of S . Second, we consider the case when $cur.next$ is **null**, which is depicted in node C'' . Node E is obtained from C'' after loading registers cur and ys onto the stack. At $pc = 14$ a **Putfield** instruction is performed. Therefore we perform a refinement according to Definition 7.3. We just illustrate two different cases, E' and E'' respectively. Nodes F' and F'' are obtained after performing the **Putfield** instruction.

The above definition of computation graph is non-constructive. First, the definition of *unifier* demands to check all reachable instances of the given state. This is clearly non-computable. However, we can always approximate the unifier by using standard unification arguments, which is precisely what we do in our implementation. Alternatively, we could employ annotations as in [27, 8, 7, 6]. Second, we have not yet clarified how widening is performed in general. We can make this step constructive by the use of the join operation defined in Definition 6.6. Whenever we are about to finish a loop, we attempt to use an instance refinement to the state starting this loop. If this fails, for example in an attempted step from B to A in Example 7.2, we widen the corresponding state. Here we collect all states that need to be abstracted and join them to obtain an abstraction. Complementing the proposed widening strategy, we restrict the applications of class instance and unsharing suitably, such that these refinement steps are only performed if no other steps are applicable.

The next lemma shows that if this strategy is followed we are guaranteed to obtain a *finite* computation graph.

Lemma 7.1. *Let G be the computation graph of a program P such that in the construction of G the above widening construction is applied whenever possible. Then G is finite.*

Proof. We argue indirectly. Suppose the computation graph G of P is infinite. This is only possible if there exists an initial state $start$ of P that is non-terminating. This implies that starting from $start$ we reach a loop in P that is called infinitely often. As G is finite this implies that the widening operation for this loop gives rise to an infinite sequence of states $(s_i)_{i \geq 0}$ such that $s_i \sqsubseteq s_{i+1}$ for all i . However, this is impossible as any ascending chain of abstract states is finite, as for all i : $\|s_i\| > \|s_{i+1}\|$. \square

Let G be a computation graph. We write $s \rightarrow_G t$ to indicate that state t is directly reachable in G from s . Sometimes we want to distinguish whether t is obtained by an abstract computation (denoted as $s \rightarrow_{\text{cmp}} t$) or by a widening step (denoted as $s \rightarrow_{\text{wid}} t$). If t is reachable from s in G we write $s \xrightarrow{*}_G t$. If $s \neq t$ this is denoted as $s \xrightarrow{+}_G t$. If at most one step was performed we write $s \xrightarrow{=}_G t$.

Lemma 7.2. *Let s be a state and let s' be a JVM state such that $s' \sqsubseteq s$. Then $P: s' \xrightarrow{\text{jvm}}_1 t'$ implies the existence of a state t such that $t' \sqsubseteq t$ and $s \xrightarrow{*}_{\text{cmp}} \cdot \xrightarrow{*}_{\text{cmp}} \cdot \xrightarrow{=}_{\text{wid}} t$.*

Proof. In proof, we proceed by case distinction on the instruction executed to perform $P: s' \xrightarrow{\text{jvm}}_1 t'$. This will show the existence of state t such that $t' \sqsubseteq t$ where $s \xrightarrow{*}_{\text{cmp}} \cdot \xrightarrow{*}_{\text{cmp}} t$ holds. More precisely at most two refinement steps may be necessary before the JVM instruction is symbolically evaluated in G . In order to reach from t a widened

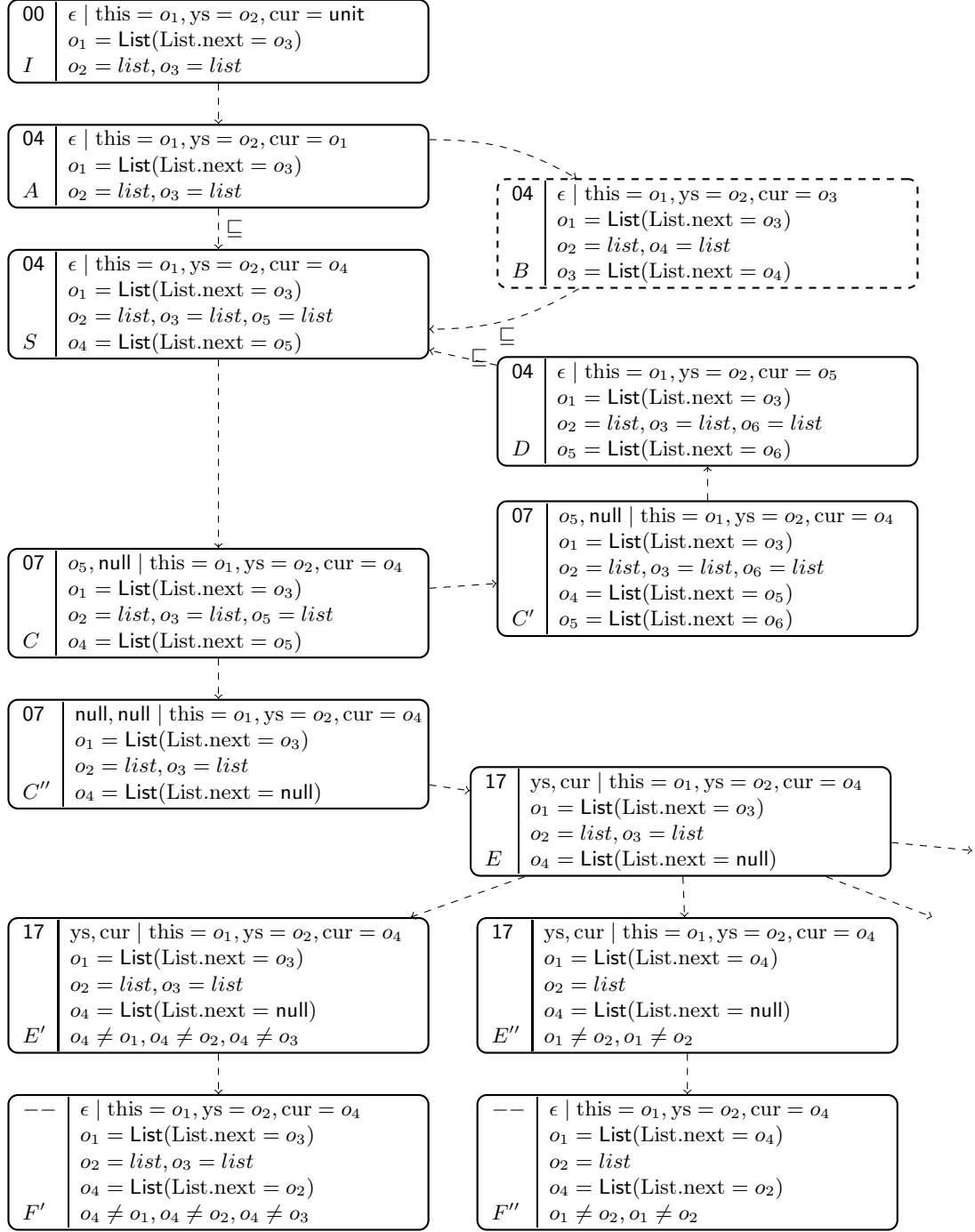


Figure 10: The (incomplete) computation graph of List.append.

state in G an instance edge may be necessary. This explains the optional widening step proposed in the lemma.

By assumption $s' \sqsubseteq s$, let $s' = (heap', frm' :: frms', iu')$ and $s = (heap, frm :: frms, iu)$. We only treat some informative cases and employ the notation from Figure 8.

- Consider **Load** n . By definition of \sqsubseteq the number of registers in frm' and frm coincide. In particular $loc(n)$ is defined and its value can be symbolically loaded to obtain an abstract state t such that $t' \sqsubseteq t$ holds.
- Consider **Store** n . By definition of \sqsubseteq the number of stack elements in frm' and frm coincide. Furthermore the number of register coincides. Hence the instruction is executed symbolically to obtain a state t such that $t' \sqsubseteq t$ holds.
- Consider **Push** v . The instruction can be directly executed, as v is a concrete value.
- Consider **Pop**. Similarly to the **Push** instruction, but the value v may now be an abstract value.
- Consider **New** cn' . Instead of creating a new object of class cn' a new abstract variable of type cn' is created.
- Consider **Getfield** $fn\ cn'$. By definition of \sqsubseteq the number of stack elements in frm' and frm coincide. Furthermore a' is an address in state s' . Hence the corresponding element a on the stack in state s is an address, too. However, the value $heap(a)$ may be abstract and need not contain the field fn . In order to create this field, a class instance refinements needs to be invoked. After this refinement, the instruction can be executed.
- Consider **Putfield** $fn\ cn'$. Analogous to the case of **Getfield**, but the crucial difference to the **Getfield** instruction is that **Putfield** can only be symbolically executed, if fn is only reachable via the address a . This guarantees $t' \sqsubseteq t$ but may require additional unsharing refinement steps.
- Consider **Checkcast** cn' . By assumption on P the cast check is void. Hence the (abstract) state remains unchanged.
- Consider **Invoke** $mn'\ n$. By definition of \sqsubseteq the number of stack elements in frm' and frm coincide. Hence method mn' can be directly invoked in the abstract state s . Let a , and p_0, \dots, p_{n-1} denote the address of the calling object and p_0, \dots, p_{n-1} the parameters in s . Then the top-frame in the resulting state t has the following form:

$$frm'' = ([, [a, p_0, \dots, p_{n-1}] @ units, cn'', mn', 0) ,$$

where cn'' is suitable defined. As the address a and the p_0, \dots, p_{n-1} are generalisations of the corresponding address and parameters in s' we obtain $t' \sqsubseteq t$.

- Consider **Return**. Without loss of generality, we assume that the frame stack contains at least two elements. Let frm'_1 and frm'_2 denote the second frame in s' and the resulting transformation of this frame in t' , respectively. Furthermore let frm_1 denote the generalisation of frm'_1 in the abstract state s . By definition of \sqsubseteq

the number of stack elements in frm'_1 and frm_1 coincide. Hence the execution can be performed symbolically so that the frame frm_2 is obtained as a transformation of frm_1 . We obtain $frm'_2 \sqsubseteq frm_2$.

- Consider **IAdd**. Let i_2, i_1 denote the first two stack elements of s . By definition of the symbolic execution of **IAdd** we perform the step by introducing a new abstract integer i_3 and adding the constraint $i_3 = i_1 + i_2$. The thus obtained state t is more general than the concrete state t' as for any number z and any abstract integer i_3 we have $z \sqsubseteq i_3$.
- Consider **IfFalse** i . Without loss of generality let **false** denote the first element on the stack of s . Executing the symbolic step yields a state t , which is a generalisation of t' by assumption on s' and s .
- Consider **CmpEq**. The case is similar to the symbolic step **IAdd**.
- Consider **BAnd**. The case is similar to the symbolic step **IAdd**.
- Consider **Goto** i . As the state is essentially unchanged for the **Goto** instruction, the lemma follows immediately.

□

We arrive at the main result of this section.

Theorem 7.1. *Let s' and t' be JVM states, where s' is reachable from some initial state $start$. Suppose $P: s' \xrightarrow{\text{jvm}} t'$, where the runtime of the execution is m . Let G denote the computation graph of P . Then there exists an abstraction s of s' and an abstraction t of t' such that $s \xrightarrow{*}_G t$ holds. Moreover let m' denote the length of the path from s to t in G . Then $m \leq m' \leq 4m$.*

Proof. By assumption we have $P: start \xrightarrow{\text{jvm}} s'$ for some initial state $start$ of P . By definition of G there exists an abstract state $I \in G$ such that $start \sqsubseteq I$. By induction on the number of evaluation steps from $start$ to s' in conjunction with Lemma 7.2 we conclude the existence of a state $s \in G$ such that $s' \sqsubseteq s$.

By induction on m (again employing Lemma 7.2), we conclude the existence of states s and t such that $s \xrightarrow{*}_G t$. Hence, the first part of the theorem follows. Furthermore by Lemma 7.2 we directly obtain that $m \leq m'$. However, by the proof of Lemma 7.2 we see that each evaluation of the JVM is simulated by a path of length 4 in G . □

The above theorem cannot be directly employed to show that the transformation to computation graphs is non-termination preserving, as we have reasoned about a given (finite) computation in P . However, the latter follows easily by an indirect argument.

Corollary 7.1. *The transformation to computation graphs is non-termination preserving.*

Proof. Suppose there exists an infinite run in P , but any path in G is finite. Let $start$ be some initial state $start$ of P . By Theorem 7.1 there exists a state t' such that $P: start \xrightarrow{jvm} t'$ and a node $t \in G$ with $t' \sqsubseteq t$. Furthermore, as all paths in G are finite, we can assume t is the last node of such a path. However, as t' is non-terminating, there exists a successor, but by assumption there is no successor of t in G . This contradicts Lemma 7.2. \square

8. Constrained Rewrite Systems

Let G be the computation graph for program P with initial state I ; G is kept fixed for the remainder of the section. In the following we describe the translation from G into a *constrained term rewrite system* (*cTRS* for short). Our definition is a variation of cTRSs as for example defined by Falke and Kapur [9, 10] or Sakata et al. [30]. The here proposed transformation is inspired by [27]. Otto et al. transform *termination graphs* into *integer term rewrite systems* (*ITRSs* for short) [11]. In contrast to this translation, our representation of states is technically simpler, as our abstract states allow a simple graph-based representation (cf. Definition 6.2).

Let \mathcal{C} be a (not necessarily finite) sorted signature, let \mathcal{V}' denote a countably infinite set of sorted variables. Furthermore let T denote a theory over \mathcal{C} . Quantifier-free formulas over \mathcal{C} are called *constraints*. Suppose \mathcal{F} is a sorted signature that extends \mathcal{C} and let $\mathcal{V} \supseteq \mathcal{V}'$ denote an extension of the variables in \mathcal{V}' . Let $\mathcal{T}(\mathcal{F}, \mathcal{V})$ denote the set of (*sorted*) *terms* over the signature \mathcal{F} and \mathcal{V} . Note that the sorted signature is necessary to distinguish between *theory* variables that are to be interpreted over the theory T and *term* variables whose interpretation is free. A *constrained rewrite rule*, denoted as $l \rightarrow r \llbracket C \rrbracket$, is a triple consisting of terms l and r , together with a constraint C . We assert that $l \notin \mathcal{V}$, but do *not* require that $\text{Var}(l) \supseteq \text{Var}(r) \cup \text{Var}(C)$, where $\text{Var}(t)$ ($\text{Var}(C)$) denotes the variables occurring in the term t (constraint C). A *constrained term rewrite system* (*cTRS*) is a finite set of constrained rewrite rules. The proposed notion of cTRSs is inspired by [9] and in turn influenced [19].

Let \mathcal{R} denote a cTRS. A context D is a term with exactly one occurrence of a *hole* \square , and $D[t]$ denotes the term obtained by replacing the hole \square in D by the term t . A substitution σ is a function that maps variables to terms, and $t\sigma$ denotes the homomorphic extension of this function to terms. We define the rewrite relation $\rightarrow_{\mathcal{R}}$ as follows. For terms s and t , $s \rightarrow_{\mathcal{R}} t$ holds, if there exists a context D , a substitution σ and a constrained rule $l \rightarrow r \llbracket C \rrbracket \in \mathcal{R}$ such that $s = D[l\sigma]$ and $t = D[r\sigma]$ with $T \vdash C\sigma$. For extra variables x possible occurring in t we demand that (i) $\sigma(x)$ is in normal-form and (ii) that $|\sigma(x)|$ is bounded by $|l\sigma| + |r'\sigma|$, where r' is obtained from r by replacing all extra variables with the constant \square . Here $|t|$ denotes a suitable measure of the term complexity of t . We fix a specific measure below. Note that condition (i) is essential to ensure termination, while condition (ii) is essential to guarantee that the rewrite relation is finitely-branching.

We often drop the reference to the cTRS \mathcal{R} , if no confusion can arise from this. A function symbol in \mathcal{F} is called *defined* if f occurs as the root symbol of l , where $l \rightarrow r \llbracket C \rrbracket \in \mathcal{R}$.

\mathcal{R} . Function symbols in $\mathcal{F} \setminus \mathcal{C}$ that are not defined, are called *constructor* symbols, and the symbols in \mathcal{C} are called *theory* symbols.

A cTRS \mathcal{R} is called *terminating*, if the relation $\rightarrow_{\mathcal{R}}$ is well-founded. For a terminating cTRS \mathcal{R} , we define its *runtime complexity*, denoted as rctrs . We adapt the runtime complexity with respect to a standard TRS suitable for cTRS \mathcal{R} . (See [15] for the standard definition.) The *derivation height* of a term t (with respect to \mathcal{R}) is defined as the maximal length of a derivation (with respect to \mathcal{R}) starting in t . The derivation height of t is denoted as $\text{dh}(t)$.

Definition 8.1. We define the *runtime complexity* (with respect to \mathcal{R}) as follows:

$$\text{rctrs}(n) := \max\{\text{dh}(t) \mid t \text{ is basic and } |t| \leq n\},$$

where a term $t = f(t_1, \dots, t_k)$ is called *basic* if f is defined, and the terms t_i are only built over constructor, theory symbols, and variables.

In the following we are only interested in cTRS over a specific theory T , namely Presburger arithmetic, that is, we have $T \vdash C$, if all ground instances of the constraint C are valid in Presburger arithmetic. Recall, that Presburger arithmetic is decidable. If $T \vdash C$, then C is *valid*. On the other hand, if there exists a substitution σ , such that $T \vdash C\sigma$, then C is *satisfiable*.

To represent the basic operations in the Jinja bytecode instruction set (cf. Figure 3) we collect the following connectives and truth constants in \mathcal{C} : $\wedge, \vee, \neg, \text{true}$, and false , together with the following relations and operations: $=, \neq, \geq, +, -$. Furthermore, we add infinitely many constants to represent integers. We often write $l \rightarrow r$ instead of $l \rightarrow r \llbracket \text{true} \rrbracket$. As expected \mathcal{C} makes use of two sorts: **bool** and **int**. We suppose that all abstract variables X_1, X_2, \dots are present in the set of variables \mathcal{V} , where abstract integer (Boolean) variables are assigned sort **int** (**bool**) and all other variables are assigned sort **univ**. The remaining elements of the signature \mathcal{F} will be defined in the course of this section. As the signature of these function symbols is easily read off from the translation given below, in the following the sort information is left implicit, to simplify the presentation.

The size of a term t , denoted as $|t|$ is defined as follows:

$$|t| := \begin{cases} 1 & \text{if } t \text{ is a variable} \\ \text{abs}(t) & \text{if } t \text{ is an integer} \\ 1 + \sum_{i=1}^n |t_i| & \text{if } t = f(t_1, \dots, t_n) \text{ and } f \text{ is not an integer.} \end{cases}$$

The next definition embodies the fact that only tree-shaped objects can be represented as terms.

Definition 8.2. Let $s = (\text{heap}, \text{frms}, \text{iu})$ be a state reachable from the initial state I in G and let $a \in \text{dom}(\text{heap})$. Suppose there is a JVM state $t = (\text{heap}', \text{frms}')$ reachable in P from some initial state start and a morphism $m: s \rightarrow t$. We call a *special*, if one of the following conditions is fulfilled:

1. either $m(a) \stackrel{\pm}{=} m(a)$, that is $m(a)$ is cyclic in t , or
2. suppose $\text{heap}'(m(a)) = (cn, ftable)$ such that $ftable((cn, id)) = ftable((cn', id'))$ for two distinct pairs (cn, id) , (cn', id') , that is the object $\text{heap}(m(a))$ is not tree-shaped.

In the next definition, we show how a state becomes representable as term over \mathcal{F} .

Definition 8.3. Let $s = (\text{heap}, \text{frms}, iu)$ be a state and let the index sets Stk and Loc be defined as above. Suppose v is a value. Then the value v is translated as follows:

$$\text{tval}(v) := \begin{cases} \text{null} & \text{if } v \in \{\text{unit}, \text{null}\} \\ v & \text{if } v \text{ is a non-address value, except } \text{unit} \text{ or } \text{null} \\ \text{taddr}(v) & \text{if } v \text{ is an address.} \end{cases}$$

Let a be an address. Then a is translated as follows:

$$\text{taddr}(a) := \begin{cases} x & \text{if } a \text{ is special (cf. Definition 8.2) and } x \\ & \text{is a fresh variable} \\ x & \text{if } \text{heap}(a) \text{ denotes an abstract variable } x \\ cn(\text{tval}(v_1), \dots, \text{tval}(v_n)) & \text{if } \text{heap}(a) = (cn, ftable) . \end{cases}$$

Here we suppose in the last case that $\text{dom}(ftable) = \{(cn_1, id_1), \dots, (cn_n, id_n)\}$ and for all $1 \leq i \leq n$: $ftable((cn_i, id_i)) = v_i$. Finally, to translate the state s into a term, it suffices to translate the values of the registers and the operand stacks of all frames in the list frms . Let $(stk, i, j) \in Stk$ such that $stk_i(j)$ denotes the j^{th} value in the operation stack of the i^{th} frame in frms . Similarly for $(loc, i', j') \in Loc$. Then we set

$$\text{ts}(s) = [\text{tval}(stk_1(1)), \dots, \text{tval}(stk_k(|stk_k|)), \text{tval}(loc_1(1)), \dots, \text{tval}(loc_k(|loc_k|))] ,$$

where the list $[\dots]$, is formalised by an auxiliary binary symbol $::$ and the constant nil .

Example 8.1 (continued from Example 7.2). Consider the simplified presentation of state C in Figure 10. Then $\text{ts}(C)$ yields following term:

$$[list, \text{null}, \text{List}(list), list, \text{List}(\text{List}(list))] .$$

Note that we can omit the information of the defining classes of the fields, since this is already captured in the symbolic evaluation. Furthermore, observe that our term representation can only fully represent tree-shaped data structures. In this sense, the term representation of a state s is less general, than its graph-based representation. However, we still obtain the following lemma.

Lemma 8.1. Let s and t be states. If $t \sqsubseteq s$, then there exists a substitution σ such that $\text{ts}(t) = \text{ts}(s)\sigma$.

Proof. Let S and T be the state graphs of s and t , respectively. By assumption there exists a morphism $m: S \rightarrow T$. The lemma is a direct consequence of the following two observations:

- Consider the terms $\text{ts}(s)$ and $\text{ts}(t)$. By definition these terms encode the standard term representations of the graphs S and T .
- Let u and v be nodes in S and T such that $m(u) = v$. The label of u (in S) can only be distinct from the label of v (in T), if $L_S(u)$ is an abstract variable or **null**. In the former case $\text{tval}(L_S(u))$ is again a variable and the latter case implies that $L_T(v) = \text{unit}$. Thus in both cases, $\text{tval}(L_S(u))$ matches $\text{tval}(L_T(v))$.

□

The next lemma relates the size of a state to its term representation and vice versa.

Lemma 8.2. *Let $s = (\text{heap}, \text{frms}, iu)$ be state such that heap does not admit special references. Then $|\text{ts}(s)| = \|s\|$.*

Proof. As a consequence of Definition 6.3 and the above proposed variant of the term complexity we obtain $|\text{ts}(s)| \leq \|s\|$ for all states s . For the other direction observe that in the absence of special references no information is lost in the term representation and thus $\|s\| \leq |\text{ts}(s)|$. □

Definition 8.4. Let $s = (\text{heap}, \text{frms}, iu)$ be a state reachable from I and let p, q denote distinct addresses in heap . Suppose the current instruction in s is a **Putfield** instruction that alters address p and $\text{heap}(q)$ is an abstract variable for some class. Furthermore, suppose there is a JVM state $t = (\text{heap}', \text{frms}')$ reachable in P from start and a morphism $m: s \rightarrow t$. Then we say that p and q are *joinable* (denoted as $p \bowtie q$) if $m(p) \stackrel{+}{\mapsto} a \stackrel{*}{\mapsto} m(q)$ for some address $a \in \text{dom}(\text{heap}')$.

Let G be a computation graph. For any state s in G we introduce a new function symbol f_s . Suppose $\text{ts}(s) = [s_1, \dots, s_n]$. To ease presentation we write $f_s(\text{ts}(s))$ instead of $f_s(s_1, \dots, s_n)$.

Definition 8.5. Let G be a computation graph and let s and t be states in G . We define the constrained rule *corresponding* to the edge (s, t) (denoted as $\text{rule}(s, t)$) as follows:

$$\text{rule}(s, t) = \begin{cases} f_s(\text{ts}(s)) \rightarrow f_t(\text{ts}(s)) & \text{if } s \sqsubseteq t \text{ (cf. Definition 5.6)} \\ f_s(\text{ts}(t)) \rightarrow f_t(\text{ts}(t)) & \text{if } t \text{ is a class instance or unsharing refinement of } s \text{ (cf. Definitions 7.2 and 7.3)} \\ f_s(\text{ts}(s)) \rightarrow f_t(\text{ts}(t)) \llbracket \text{tval}(C) \rrbracket & \text{the edge between } s \text{ and } t \text{ is labelled by } C \\ f_s(\text{ts}(s)) \rightarrow f_t(\text{ts}^*(t)) & s \text{ corresponds to a Putfield instruction on the address } p \text{ and there exists an address } q \text{ in } s, \text{ such that } p \bowtie q \text{ (cf. Definition 8.4)} \\ f_s(\text{ts}(s)) \rightarrow f_t(\text{ts}(t)) & \text{otherwise} \end{cases}$$

Here $\text{tval}(C)$ denotes the standard extension of the mapping tval to labels of edges and ts^* is defined as ts but employs fresh variables for any reference q .

Example 8.2 (continued from Example 6.2). Consider Figure 1 and Figure 10 from Example 6.2. We use following conventions: List variables are denoted by l , followed by a number. The function symbols contain a state from the computation graph as well as the corresponding program position from the bytecode. The translation results into following cTRS rules:

Lemma 8.3. *Let s and t be states in G connected by an edge $s \xrightarrow{\ell} t$ from s to t . Suppose s' is a JVM state with $s' \sqsubseteq s$. Suppose further that if the constraint ℓ labelling the edge is non-empty, then s' satisfies ℓ . Moreover, if $s \xrightarrow{\ell} t$ follows due to a refinement step, then s' is consistent with the chosen refinement. Then there exists a JVM state $t' \sqsubseteq t$, such that $f_s(\text{ts}(s')) \rightarrow_{\text{rule}(s,t)} f_t(\text{ts}(t'))$.*

Proof. The proof proceeds by case analysis on the edge $s \xrightarrow{\ell} t$ in G , where we only need to consider the following four cases. The argument for the omitted fifth case is very similar to the third case.

- *Case $s \xrightarrow{\ell} t$, as $s \sqsubseteq t$; $\ell = \emptyset$.* By assumption $s' \sqsubseteq s \sqsubseteq t$. Hence $s' \sqsubseteq t$ by transitivity of the instance relation. By Lemma 8.1 there exists a substitution σ , such that $\text{ts}(s') = \text{ts}(s)\sigma$. In sum, we obtain:

$$f_s(\text{ts}(s')) = f_s(\text{ts}(s))\sigma \rightarrow_{\text{rule}(s,t)} f_t(\text{ts}(s))\sigma = f_t(\text{ts}(t')) ,$$

where we set $t' := s'$.

- *Case $s \xrightarrow{\ell} t$, as t is a refinement of s ; $\ell = \emptyset$.* By assumption $s' \sqsubseteq s$ and s' is concrete. Hence $s' \sqsubseteq t$ by definition of t . Again by Lemma 8.1 there exists a substitution σ , such that $\text{ts}(s') = \text{ts}(t)\sigma$. In sum, we obtain:

$$f_s(\text{ts}(s')) = f_s(\text{ts}(t))\sigma \rightarrow_{\text{rule}(s,t)} f_t(\text{ts}(t))\sigma = f_t(\text{ts}(t')) ,$$

where we again set $t' := s'$.

- *Case $s \xrightarrow{\ell} t$, as t is the result of the symbolic evaluation of s and $\ell = C \neq \emptyset$.* By assumption s' satisfies the constraint C . More precisely, there exists a substitution σ , such that $\text{ts}(s') = \text{ts}(s)\sigma$ and $T \vdash \text{tval}(C)\sigma$. We obtain:

$$f_s(\text{ts}(s')) = f_s(\text{ts}(s))\sigma \rightarrow_{\text{rule}(s,t)} f_t(\text{ts}(t))\sigma .$$

Let t' be defined such that $P: s' \xrightarrow{\text{jvm}}_1 t'$. By Lemma 7.2 we obtain $t' \sqsubseteq t$ and by inspection of the proof of Lemma 7.2 we observe that $\text{ts}(t') = \text{ts}(t)\sigma$. In sum $f_s(\text{ts}(s')) \rightarrow_{\text{rule}(s,t)} f_t(\text{ts}(t'))$.

- *Case $s \xrightarrow{\ell} t$, as t is the result of a `Putfield` instruction on p and there exists an address q in s with $p \bowtie q$.* By assumption $s' \sqsubseteq s$ and thus $\text{ts}(s') = \text{ts}(s)\sigma$ for some substitution σ . Let t' be defined such that $P: s' \xrightarrow{\text{jvm}}_1 t'$. Due to Lemma 7.2, we have $t' \sqsubseteq t$ and thus there exists a substitution τ such that $\text{ts}(t') = \text{ts}^*(t)\tau$.

Consider the rule $f_s(\text{ts}(s)) \rightarrow f_t(\text{ts}^*(t))$. By definition the address q points in s to an abstract variable x such that x occurs in $\text{ts}(s)$ and $\text{ts}(t)$. Furthermore x is replaced

$$\begin{aligned}
& f_{I00}([\text{null}, \text{null}, \text{List}(l0), l1, \text{null}]) \rightarrow f_{I01}([\text{List}(l0), \text{null}, \text{List}(l0), l1, \text{null}]) \\
& f_{I01}([\text{List}(l0), \text{null}, \text{List}(l0), l1, \text{null}]) \rightarrow f_{I02}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(l0)]) \\
& f_{I02}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(l0)]) \rightarrow f_{I03}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(l0)]) \\
& f_{I03}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(l0)]) \rightarrow f_{A04}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(l0)]) \\
& f_{A04}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(l0)]) \rightarrow f_{S04}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(l0)]) \\
& f_{S04}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(l2)]) \rightarrow f_{S05}([\text{List}(l2), \text{null}, \text{List}(l0), l1, \text{List}(l2)]) \\
& f_{S05}([\text{List}(l2), \text{null}, \text{List}(l0), l1, \text{List}(l2)]) \rightarrow f_{S06}([l2, \text{null}, \text{List}(l0), l1, \text{List}(l2)]) \\
& f_{S06}([l2, \text{null}, \text{List}(l0), l1, \text{List}(l2)]) \rightarrow f_{C07}([l2, \text{null}, \text{List}(l0), l1, \text{List}(l2)]) \\
& f_{C07}([\text{List}(l3), \text{null}, \text{List}(l0), l1, \text{List}(\text{List}(l3))]) \rightarrow f_{C'07}([\text{List}(l3), \text{null}, \text{List}(l0), l1, \text{List}(\text{List}(l3))]) \\
& f_{C'07}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(\text{null})]) \rightarrow f_{C''07}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(\text{null})]) \\
& f_{C'07}([\text{List}(l3), \text{null}, \text{List}(l0), l1, \text{List}(\text{List}(l3))]) \rightarrow f_{C'08}([\text{true}, \text{null}, \text{List}(l0), l1, \text{List}(\text{List}(l3))]) \\
& f_{C'08}([\text{true}, \text{null}, \text{List}(l0), l1, \text{List}(\text{List}(l3))]) \rightarrow f_{C'09}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(\text{List}(l3))]) \\
& f_{C'09}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(\text{List}(l3))]) \rightarrow f_{C'10}([\text{List}(\text{List}(l3)), \text{null}, \text{List}(l0), l1, \text{List}(\text{List}(l3))]) \\
& f_{C'10}([\text{List}(\text{List}(l3)), \text{null}, \text{List}(l0), l1, \text{List}(\text{List}(l3))]) \rightarrow f_{C'11}([\text{List}(l3), \text{null}, \text{List}(l0), l1, \text{List}(\text{List}(l3))]) \\
& f_{C'11}([\text{List}(l3), \text{null}, \text{List}(l0), l1, \text{List}(\text{List}(l3))]) \rightarrow f_{C'12}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(l3)]) \\
& f_{C'12}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(l3)]) \rightarrow f_{C'13}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(l3)]) \\
& f_{C'13}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(l3)]) \rightarrow f_{C'14}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(l3)]) \\
& f_{C'14}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(l3)]) \rightarrow f_{D04}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(l3)]) \\
& f_{D04}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(l3)]) \rightarrow f_{S04}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(l3)]) \\
& f_{C''07}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(\text{null})]) \rightarrow f_{C''08}([\text{false}, \text{null}, \text{List}(l0), l1, \text{List}(\text{null})]) \\
& f_{C''08}([\text{false}, \text{null}, \text{List}(l0), l1, \text{List}(\text{null})]) \rightarrow f_{C''15}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(\text{null})]) \\
& f_{C''15}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(\text{null})]) \rightarrow f_{C''16}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(\text{null})]) \\
& f_{C''16}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(\text{null})]) \rightarrow f_{E17}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(\text{null})]) \\
& f_{E17}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(\text{null})]) \rightarrow f_{E'17}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(\text{null})]) \\
& f_{E17}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(\text{null})]) \rightarrow f_{E'17}([\text{null}, \text{null}, \text{List}(\text{List}(\text{null})), l1, \text{List}(\text{null})]) \\
& f_{E'17}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(\text{null})]) \rightarrow f_{E'8}([\text{List}(\text{null}), \text{null}, \text{List}(l0), l1, \text{List}(\text{null})]) \\
& f_{E'18}([\text{List}(\text{null}), \text{null}, \text{List}(l0), l1, \text{List}(\text{null})]) \rightarrow f_{E'19}([l1, \text{List}(\text{null}), \text{List}(l0), l1, \text{List}(\text{null})]) \\
& f_{E'19}([l1, \text{List}(\text{null}), \text{List}(l0), l1, \text{List}(\text{null})]) \rightarrow f_{E'20}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(l1)]) \\
& f_{E'20}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(l1)]) \rightarrow f_{E'21}([\text{null}, \text{null}, \text{List}(l0), l1, \text{List}(l1)]) \\
& f_{E'17}([\text{null}, \text{null}, \text{List}(\text{List}(\text{null})), l1, \text{List}(\text{null})]) \rightarrow f_{E''8}([\text{List}(\text{null}), \text{null}, \text{List}(\text{List}(\text{null})), l1, \text{List}(\text{null})]) \\
& f_{E''18}([\text{List}(\text{null}), \text{null}, \text{List}(\text{List}(\text{null})), l1, \text{List}(\text{null})]) \rightarrow f_{E''19}([l1, \text{List}(\text{null}), \text{List}(\text{List}(\text{null})), l1, \text{List}(\text{null})]) \\
& f_{E''19}([l1, \text{List}(\text{null}), \text{List}(\text{List}(\text{null})), l1, \text{List}(\text{null})]) \rightarrow f_{E''20}([\text{null}, \text{null}, \text{List}(\text{List}(l1)), l1, \text{List}(l1)]) \\
& f_{E''20}([\text{null}, \text{null}, \text{List}(\text{List}(l1)), l1, \text{List}(l1)]) \rightarrow f_{E''21}([\text{null}, \text{null}, \text{List}(\text{List}(l1)), l1, \text{List}(l1)]) \\
& \dots
\end{aligned}$$

Figure 11: The cTRS of append.

by an extra variable x' in $\text{ts}^*(t)$. To simplify the presentation, we assume that x' is the only extra variable in $\text{ts}^*(t)$. Let m be a morphism such that $m: s \rightarrow s'$. Without loss of generality, we assume that there exists an address a in s' such that $m(p) \stackrel{\perp}{\rightarrow} a \stackrel{*}{\rightarrow} m(q)$. By definition of `Putfield`, $m(p)$ and $m(q)$ exist in t' and only

the part of the heap reachable from these addresses can differ in s' and t'

In order to show the admissibility of the rewrite step $f_s(\text{ts}(s')) \rightarrow f_t(\text{ts}(t'))$ we define a substitution ρ such that $\text{ts}(s)\rho = \text{ts}(s')$ and $\text{ts}^*(t)\rho = \text{ts}(t')$. We set:

$$\rho(y) := \begin{cases} \tau(x) & \text{if } y = x' \\ \sigma(y) & \text{otherwise .} \end{cases}$$

Then $\text{ts}(s)\rho = \text{ts}(s')$ by definition as $x' \notin \text{Var}(s)$. On the other hand $\text{ts}^*(t)\rho = \text{ts}(t')$ follows as the definition of ρ forces the correct instantiation of x' and Lemma 7.2 implies that σ and τ coincide on the portion of the heap not changed by the **Putfield** instruction.

Finally, we have to verify the size condition on x' . However, by construction we have

$$|\rho(x')| = |\tau(x)| \leq |f_s(\text{ts}(s))\rho| + |f_t(\text{ts}^*(t))\{x' \mapsto \square\}\rho| .$$

□

Let G be the computation graph of P . Further, let \mathcal{R} denote the collection of rules representing G according to Definition 8.5.

Our definition of the obtained cTRS \mathcal{R} is non-constructive, as neither the definition of a special address (cf. Definition 8.2) nor the definition of joinable references (cf. Definition 8.4) are computable. This does not affect our theoretical results, but calls for a suitable approximation in the *implementation* of the transformation. For that one can either employ a general shape analysis as for example detailed in [31, 29, 25] or employ the annotation technique proposed in [27]. We use the later possibility in the implementation of our technique.

The next lemma emphasises that any execution step is represented by at least one and at most four rewrite steps with respect to \mathcal{R} .

Lemma 8.4. *Let s be a state in G and let s' be a JVM state such that $s' \sqsubseteq s$. Then $P: s' \xrightarrow{\text{jvm}}_1 t'$ implies the existence of a state $t \in G$ such that $t' \sqsubseteq t$ and $f_s(\text{ts}(s')) \xrightarrow{\leq 4} f_t(\text{ts}(t'))$. (Here $\xrightarrow{\leq m}$ denotes at least one and at most m many rewrite steps in \mathcal{R} .)*

Proof. The lemma follows from the proof of Lemma 7.2 and Lemma 8.3. □

We arrive at the main result of this paper.

Theorem 8.1. *Let s' and t' be JVM states. Suppose $P: s' \xrightarrow{\text{jvm}} t'$, where s' is reachable in P from some initial state start. Then there exists an abstraction s of s' and a derivation $f_s(\text{ts}(s')) \rightarrow_{\mathcal{R}}^+ f_t(\text{ts}(t'))$ such that $t' \sqsubseteq t$. Furthermore $\text{rcjvm}(n) \leq \text{rctrs}(n) \leq 4 \cdot \text{rcjvm}(n)$.*

Proof. Due to Theorem 7.1 there exists a s such that $s' \sqsubseteq s$. Now, let m denote the runtime of the execution $P: s' \xrightarrow{\text{jvm}} t'$. Then by induction on m in conjunction with Lemma 8.4 we obtain the existence of a state t such that $t' \sqsubseteq t$ and a derivation D :

$$f_s(\text{ts}(s')) \xrightarrow{\leq 4m} f_t(\text{ts}(t')) . \tag{1}$$

In particular we have $f_s(\text{ts}(s')) \rightarrow_{\mathcal{R}}^+ f_t(\text{ts}(t'))$ from which the first part of the theorem follows.

In order to conclude the second part, suppose m denotes the runtime of the execution $P: \text{start} \xrightarrow{\text{jvm}} t'$. As G is the computation graph of P we obtain $\text{start} \sqsubseteq I$. By our assumptions on the input data to P , Lemma 8.2 yields $|\text{ts}(\text{start})| = \|\text{start}\|$. Specialising (1) to I and start yields $f_I(\text{ts}(\text{start})) \xrightarrow{\leq 4m} f_t(\text{ts}(t'))$. Thus we obtain

$$m \leq \text{rctr}(|\text{ts}(\text{start})|) = \text{rctr}(\|\text{start}\|) \leq 4m .$$

As m was arbitrary and $m \leq \text{rcjvm}(\|\text{start}\|)$, the second part of the theorem follows. \square

Corollary 8.1. *The computation graph method, that is the transformation from a given JBC program P to a cTRS \mathcal{R} is non-termination preserving.*

Proof. Reasoning similar as in the proof of Corollary 7.1, we obtain that the proposed transformation from P to \mathcal{R} is non-termination preserving. \square

9. Implementation

A prototype of the proposed method has been implemented in the Haskell programming language.

Motivated by abstract interpretation we allow different instantiations of abstract domains. In particular we allow instances for *IntDomain* and *MemoryModel*:

Integer operations are usually performed locally, ie., independent from other components of the state. To define an instance for an abstract integer domain, arithmetic operations, a widening operator and an instance check have to be defined. One can easily provide different domains, such as the domain of intervals or the domain of signs. A memory model allows different representations of the heap abstraction. For example a sharing domain as presented here, or a distinctness domain as described initially by Otto et al. [8]. The choice of a memory model has a significant impact on state representation and state operations. Therefore it is necessary to provide all functions that access or modifies the dynamic part of the memory. States itself are represented in a natural way using algebraic data types. Most operations such as the unification algorithm or the widening operation are reduced to a map operation over the state. Though we usually have to memorise visited nodes in the heap to circumvent looping, or corresponding addresses (cf. Definition 6.6) of two different states.

The construction of the computation graph itself is independent from the chosen domains and is a variant of the worklist algorithm. Currently different paths are processed sequently, which avoids the need of synchronisation when merging multiple states. The here proposed method does not explicitly perform (non-)cyclicity analysis as needed for the translation into the rewriting system. To infer a bound for our motivating example we introduce additional annotations in the spirit of [8] for our implementation. TCT is able to infer a linear bound from the resulting rewriting system.

10. Conclusion and Future Work

In this paper we define a representation of JBC executions as *computation graphs* from which we obtain a representation of JBC executions as *constrained rewrite systems*. We precise the *widening* of abstract states so that the representation of JBC executions is provably finite. Furthermore, we show that the resulting transformation is complexity preserving by a linear factor.

As emphasised above our approach does not directly give rise to an automatable complexity-preserving transformation, but for that requires an extension by an external shape analysis. This may appear as a deficit as in principle we aim at *automatable* complexity preserving transformations. However our main result applies to *any* computable approximation of the transformation and in particular it shows complexity preservation by a linear factor of the transformation proposed by Otto et al. [27]. Moreover, it allows for an easy incorporation of the existing wealth of results on shape analysis present in the literature and thus improves upon the modularity of the proposed transformational approach. To assess the viability of our method in practise, we have implemented a suitable approximation as a prototype that can be used as a frontend of TCT^4 . Unsurprisingly this set-up can handle our simple motivating example, but it is too early to attempt a sensible experimental assessment.

For that, we crucially have to overcome the second and third obstacle mentioned in the introduction: a) methods of runtime complexity analysis for cTRSs need to be developed and b) compositionality of the analysis is required. Item a) clarifies why we have crafted the transformation such that constrained TRSs are obtained rather than integer TRS (as in the termination graph approach). Based on very recent results in [19] on the versatility of cTRSs, we expect that it will be relatively easy to establish powerful complexity analysis for cTRSs. Furthermore, compositionality of the analysis currently amounts to the question, whether it is possible to study individual cycles in the computation graph separately. Both these questions will be investigated in the future.

References

- [1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-form upper bounds in static cost analysis. *JAR*, 46(2), 2011.
- [2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *TCS*, 413(1):142–159, 2012.
- [3] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *Proc. 17th SAS*, volume 6337 of *LNCS*, pages 117–133, 2010.
- [4] M. Avanzini and G. Moser. A Combination Framework for Complexity. 2013. Submitted to RTA’13.

⁴ <http://cl-informatik.uibk.ac.at/software/tct/>.

- [5] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *Proc. 19th CAV*, volume 4590 of *LNCS*, pages 178–192, 2007.
- [6] M. Brockschmidt, R. Musiol, C. Otto, and J. Giesl. Automated Termination Proofs for Java Bytecode with Cyclic Data. In *Proc. of 24th CAV*, volume 7358 of *LNCS*, pages 105–122, 2012.
- [7] M. Brockschmidt, C. Otto, and J. Giesl. Modular termination proofs of recursive java bytecode programs by term rewriting. In *Proc. of 22nd RTA*, *LIPIcs*, pages 155–170, 2011.
- [8] M. Brockschmidt, C. Otto, C. von Essen, and J. Giesl. Termination Graphs for Java Bytecode. In *Verification, Induction, Termination Analysis*, volume 6463 of *LNCS*, pages 17–37, 2010.
- [9] S. Falke and D. Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In *Proc. of 22nd CADE*, *CADE-22*, pages 277–293, Berlin, Heidelberg, 2009. Springer Verlag.
- [10] S. Falke, D. Kapur, and C. Sinz. Termination Analysis of C Programs Using Compiler Intermediate Languages. In *Proc. of 22nd RTA*, volume 10 of *LIPIcs*, pages 41–50, 2011.
- [11] C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. Proving Termination of Integer Term Rewriting. In *Proc. of 20th RTA*, volume 5595 of *LNCS*, pages 32–47, 2009.
- [12] S. Gulwani. Speed: Symbolic complexity bound analysis. In *Proc. 21st CAV*, volume 5643 of *LNCS*, pages 51–62, 2009.
- [13] S. Gulwani, K. Mehra, and T. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *Proc. 36th POPL*, pages 127–139. ACM, 2009.
- [14] S. Gulwani and F. Zuleger. The reachability-bound problem. In *Proc. PLDI'10*, pages 292–304. ACM, 2010.
- [15] N. Hirokawa and G. Moser. Automated Complexity Analysis Based on the Dependency Pair Method. In *Proc. of 4th IJCAR*, volume 5195 of *LNCS*, pages 364–380, 2008.
- [16] N. Hirokawa and G. Moser. Complexity, graphs, and the dependency pair method. In *Proc. of 15th LPAR*, pages 652–666, 2008.
- [17] N. Hirokawa and G. Moser. Automated Complexity Analysis Based on the Dependency Pair Method. *CoRR*, abs/1102.3129, 2011. submitted.

- [18] G. Klein and T-Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.
- [19] C. Kop and N. Nishida. Term rewriting with logical constraints. submitted to RTA’13, 2013.
- [20] A. Lochbihler. Jinja with threads. *Archive of Formal Proofs*, 2007, 2007.
- [21] A. Lochbihler. Verifying a compiler for java threads. In *Proc. 19th ESOP*, volume 6012 of *LNCS*, pages 427–447, 2010.
- [22] A. Middeldorp, G. Moser, F. Neurauter, J. Waldmann, and H. Zankl. Joint spectral radius theory for automated complexity analysis of rewrite systems. In *Proc. of 4th CAI*, *LNCS*, pages 1–20, 2011.
- [23] G. Moser. Proof Theory at Work: Complexity Analysis of Term Rewrite Systems. *CoRR*, abs/0907.5527, 2009. Habilitation Thesis.
- [24] F. Nielson, H. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 2005.
- [25] D. Nikolic and F. Spoto. Reachability analysis of program variables. In *Proc. 6th IJCAR*, volume 7364 of *LNCS*, pages 423–438, 2012.
- [26] L. Noschinski, F. Emmes, and J. Giesl. A dependency pair framework for innermost complexity analysis of term rewrite systems. In *Proc. of 23rd CADE*, volume 6803 of *LNCS*, pages 422–438, 2011.
- [27] C. Otto, M. Brockschmidt, C. v. Essen, and J. Giesl. Automated termination analysis of Java bytecode by term rewriting. In *Proc. of 21th RTA*, pages 259–276, 2010.
- [28] S. E. Panitz and M. Schmidt-Schauß. Tea: Automatically proving termination of programs in a non-strict higher-order functional language. In *Proc. of 4th SAS*, pages 345–360. Springer Verlag, 1997.
- [29] S. Rossignoli and F. Spoto. Detecting non-cyclicity by abstract compilation into boolean functions. In *Proc. 7th VMCAI*, volume 3855 of *LNCS*, pages 95–110, 2006.
- [30] T. Sakata, N. Nishida, and T. Sakabe. On proving termination of constrained term rewrite systems by eliminating edges from dependency graphs. In *Proc. of 20th WFLP*, volume 6816 of *LNCS*, pages 138–155, 2011.
- [31] S. Secci and F. Spoto. Pair-sharing analysis of object-oriented programs. In *Proc. 12th SAS*, volume 3672 of *LNCS*, pages 320–335, 2005.
- [32] M. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In *Proc. ILPS’95*, pages 465–479. MIT Press, 1995.

- [33] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer Verlag, 2001.
- [34] H. Zankl and M. Korp. Modular complexity analysis via relative complexity. In *Proc. 21th RTA*, volume 6 of *LIPIcs*, pages 385–400, 2010.
- [35] F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *Proc. 18th SAS*, volume 6887 of *LNCS*, pages 280–297. Springer Verlag, 2011.

A. Semantics of Jinja Bytecode Instructions

Load n	$\frac{(heap, (stk, loc, cn, mn, pc) :: frms)}{(heap, (loc(n) :: stk, loc, cn, mn, pc + 1) :: frms)}$
Store n	$\frac{(heap, (v :: stk, loc, cn, mn, pc) :: frms)}{(heap, (stk, loc\{n \mapsto v\}, cn, mn, pc + 1) :: frms)}$
Push v	$\frac{(heap, (stk, loc, cn, mn, pc) :: frms)}{(heap, (v :: stk, loc, cn, mn, pc + 1) :: frms)}$
Pop	$\frac{(heap, (v :: stk, loc, cn, mn, pc) :: frms)}{(heap, (stk, loc, cn, mn, pc + 1) :: frms)}$
New cn'	$\frac{(heap, (stk, loc, cn, mn, pc) :: frms)}{(heap\{a \mapsto obj\}, (a :: stk, loc, cn, mn, pc + 1) :: frms)}$
Getfield $fn\ cn'$	$\frac{(heap, (a :: stk, loc, cn, mn, pc) :: frms)}{(heap, (f\text{table}(cn', fn) :: stk, loc, cn, mn, pc + 1) :: frms)}$
Putfield $fn\ cn'$	$\frac{(heap, (v :: a :: stk, loc, cn, mn, pc) :: frms)}{(heap\{a \mapsto (cn'', f\text{table}')\}, (stk, loc, cn, mn, pc + 1) :: frms)}$
Checkcast cn'	$\frac{(heap, (v :: stk, loc, cn, mn, pc) :: frms)}{(heap, (v :: stk, loc, cn, mn, pc + 1) :: frms)}$
Invoke $mn'\ n$	$\frac{(heap, (p_{n-1} :: \dots :: p_0 :: a :: stk, loc, cn, mn, pc) :: frms)}{(heap, frm' :: (p_{n-1} :: \dots :: p_0 :: a :: stk, loc, cn, mn, pc) :: frms)}$
Return	$\frac{(heap, [])}{(heap, frm' :: frms)}$
IAdd	$\frac{(heap, (i_2 :: i_1 :: stk, loc, cn, mn, pc) :: frms)}{(heap, ((i_2 + i_1) :: stk, loc, cn, mn, pc + 1) :: frms)}$
IfFalse i	$\frac{(heap, (false :: stk, loc, cn, mn, pc) :: frms)}{(heap, (stk, loc, cn, mn, pc + i) :: frms)}$
	$\frac{(heap, (true :: stk, loc, cn, mn, pc) :: frms)}{(heap, (stk, loc, cn, mn, pc + 1) :: frms)}$
CmpEq	$\frac{(heap, (v_2 :: v_1 :: stk, loc, cn, mn, pc) :: frms)}{(heap, ((v_2 = v_1) :: stk, loc, cn, mn, pc + 1) :: frms)}$
Goto i	$\frac{(heap, (stk, loc, cn, mn, pc) :: frms)}{(heap, (stk, loc, cn, mn, pc + i) :: frms)}$
ISub	$\frac{(heap, (i_2 :: i_1 :: stk, loc, cn, mn, pc) :: frms)}{(heap, ((i_2 - i_1) :: stk, loc, cn, mn, pc + 1) :: frms)}$
CmpNeq	$\frac{(heap, (v_2 :: v_1 :: stk, loc, cn, mn, pc) :: frms)}{(heap, ((v_2 \neq v_1) :: stk, loc, cn, mn, pc + 1) :: frms)}$
CmpGeq	$\frac{(heap, (v_2 :: v_1 :: stk, loc, cn, mn, pc) :: frms)}{(heap, ((v_2 \geq v_1) :: stk, loc, cn, mn, pc + 1) :: frms)}$
BAnd	$\frac{(heap, (b_2 :: b_1 :: stk, loc, cn, mn, pc) :: frms)}{(heap, ((b_2 \wedge b_1) :: stk, loc, cn, mn, pc + 1) :: frms)}$
BOr	$\frac{(heap, (b_2 :: b_1 :: stk, loc, cn, mn, pc) :: frms)}{(heap, ((b_2 \vee b_1) :: stk, loc, cn, mn, pc + 1) :: frms)}$
BNot	$\frac{(heap, (b :: stk, loc, cn, mn, pc) :: frms)}{(heap, ((\neg b) :: stk, loc, cn, mn, pc + 1) :: frms)}$